

LINUX 应用 程序开发指南

——使用 GTK + GNOME 库

目 录

第一部分 Linux GUI 编程框架及编程基础

第 1 章 Linux 软件开发概述 1

- 1.1 关于 Linux 1
- 1.2 关于 Linux 的桌面环境 2
- 1.3 Linux 系统中的软件开发 3
 - 1.3.1 开发所使用的库 3
 - 1.3.2 Gnome 的开发结构 4
- 1.4 开发 Linux 应用程序的编程语言和编程工具 6
- 1.5 本书的结构 7

第 2 章 Gtk+/Gnome 开发简介 8

- 2.1 安装 Gtk+/Gnome 库 8
- 2.2 第一个 Gtk+应用程序 9
 - 2.2.1 一个什么也不能做的窗口 9
 - 2.2.2 示例代码的含义 9
 - 2.2.3 GTK 的 Hello World 10
 - 2.2.4 Gtk+的信号和回调函数原理 12
 - 2.2.5 Hello World 代码解释 14
 - 2.2.6 运行 helloworld 17
- 2.3 Gnome 应用程序 17
- 2.4 GNU C 编译器 18
 - 2.4.1 使用 gcc 18
 - 2.4.2 gcc 选项 18

2.5	初始化库	19
2.6	用 popt 分析参数	20
2.6.1	参数分析方法	20
2.6.2	GnomeHello 程序的参数分析	22
2.7	国际化	25
2.8	保存配置信息	27
2.8.1	读出存储的配置数据	28
2.8.2	在配置文件中存储数据	30
2.8.3	配置文件迭代器	30
2.8.4	节迭代器	33
2.8.5	其他的配置文件操作	33
2.9	会话管理	34
2.10	Gtk+的主循环	36
2.10.1	主循环基本知识	36
2.10.2	退出函数	36
2.10.3	Timeout 函数	37
2.10.4	idle 函数	37
2.10.5	输入函数	38
2.11	编译应用程序	39
2.11.1	生成源代码树	39
2.11.2	configure.in 文件	41
2.11.3	Makefile.am 文件	43
2.11.4	安装支持文件	44

第二部分 Linux 编程常用 C 语言函数库及构件库

第 3 章 glib 库简介 49

3.1	类型定义	49
3.2	glib 的宏	49
3.2.1	常用宏	49
3.2.2	调试宏	50
3.3	内存管理	52
3.4	字符串处理	53
3.5	数据结构	55
3.5.1	链表	55
3.5.2	树	59
3.5.3	哈希表	63

- 3.6 GString 65
- 3.7 计时器函数 66
- 3.8 错误处理函数 67
- 3.9 其他实用函数 67

第 4 章 构件定位 69

- 4.1 构件的显现、映射和显示 69
- 4.2 其他的构件概念 70
- 4.3 构件的类型转换 72
- 4.4 组装构件 72
 - 4.4.1 尺寸分配 73
 - 4.4.2 GtkWindow 构件 74
 - 4.4.3 GtkBox 76
 - 4.4.4 表格构件 GtkTable 79
 - 4.4.5 固定容器构件 GtkFixed 83
 - 4.4.6 布局容器构件 GtkLayout 85

第 5 章 按钮构件 87

- 5.1 普通按钮 GtkButton 87
- 5.2 开关按钮 GtkToggleButton 90
- 5.3 检查按钮 GtkCheckButton 91
- 5.4 无线按钮 GtkRadioButton 91

第 6 章 调整对象 95

- 6.1 创建一个调整对象 95
- 6.2 使用调整对象 95
- 6.3 调整对象内部机制 96

第 7 章 文本构件 GtkText 98

- 7.1 创建、配置文本构件 98
- 7.2 操作文本 99
- 7.3 键盘快捷键 100
- 7.4 GtkText 示例 100

第 8 章 范围构件 GtkRange 105

- 8.1 滚动条构件 GtkScrollBar 105
- 8.2 比例构件 GtkScale 105
 - 8.2.1 函数和信号 105

- 8.2.2 常用的范围函数 106
- 8.2.3 键盘和鼠标绑定 107
- 8.2.4 示例 107

第 9 章 杂项构件 114

- 9.1 标签构件 GtkLabel 114
- 9.2 箭头构件 GtkArrow 117
- 9.3 工具提示对象 GtkTooltips 119
- 9.4 进度条构件 GtkProgressBar 120
- 9.5 对话框构件 126
- 9.6 pixmap 127
- 9.7 标尺构件 GtkRuler 134
- 9.8 文本输入构件 GtkEntry 137
- 9.9 微调按钮构件 GtkSpinButton 140
- 9.10 组合框 GtkCombo 146
- 9.11 日历构件 GtkCalendar 148
- 9.12 颜色选择构件 GtkColorSelect 158
- 9.13 文件选择构件 GtkFileSelect 162

第 10 章 容器构件 GtkContainer 165

- 10.1 事件盒构件 GtkEventBox 165
- 10.2 对齐构件 GtkAlignment 166
- 10.3 框架构件 GtkFrame 167
- 10.4 比例框架构件 GtkAspectFrame 169
- 10.5 分栏窗口构件 GtkPanedWindow 170
- 10.6 视角构件 GtkViewport 174
- 10.7 滚动窗口构件 GtkScrolledWindow 175
- 10.8 按钮盒构件 GtkButtonBox 177
- 10.9 工具条构件 GtkToolbar 181
- 10.10 笔记本构件 GtkNotebook 187

第 11 章 分栏列表构件 GtkCList 193

- 11.1 创建分栏列表构件 GtkCList 193
- 11.2 操作模式 193
- 11.3 操作分栏列表构件列标题 194
- 11.4 操纵列表 194
- 11.5 向列表中添加行 196
- 11.6 在单元格中设置文本和 pixmap 图片 197

- 11.7 存储数据指针 198
- 11.8 处理选择 198
- 11.9 信号 199
- 11.10 GtkCList 示例 199

第 12 章 树构件 204

- 12.1 创建新树构件 204
 - 12.1.1 添加一个子树 204
 - 12.1.2 处理选中的列表 205
 - 12.1.3 树构件内部机制 205
 - 12.1.4 信号 206
 - 12.1.5 函数和宏 206
- 12.2 树项构件 GtkTreeItem 208
 - 12.2.1 信号 209
 - 12.2.2 函数和宏 210
- 12.3 树构件示例 210

第 13 章 GnomeApp 构件和 GnomeUI Info 215

- 13.1 主窗口 GnomeApp 215
- 13.2 GnomeUIInfo 216
 - 13.2.1 创建 GnomeUIInfo 216
 - 13.2.2 将 GnomeUIInfo 转换为构件 218

第 14 章 状态条构件 221

- 14.1 状态条构件简介 221
- 14.2 GnomeAppBar 构件 221
- 14.3 状态条构件 GtkStatusbar 222

第 15 章 对话框 225

- 15.1 GnomeDialog 构件 225
 - 15.1.1 创建对话框 225
 - 15.1.2 填充对话框 226
 - 15.1.3 处理 GnomeDialog 的信号 226
 - 15.1.4 最后的修饰 227
- 15.2 模态对话框 229
- 15.3 一个对话框示例 230
- 15.4 特殊对话框 231
 - 15.4.1 GnomeAbout 231

- 15.4.2 GnomePropertyBox—属性框 233
- 15.4.3 GnomeMessageBox—消息框 234

第 16 章 GDK 基础 236

- 16.1 GDK 和 Xlib 236
- 16.2 GdkWindow 237
 - 16.2.1 GdkWindow 和 GtkWidget 237
 - 16.2.2 GdkWindow 属性 238
- 16.3 视件和颜色表 240
 - 16.3.1 GdkVisual 240
 - 16.3.2 视件的类型 241
 - 16.3.3 颜色和 GdkColormap 242
 - 16.3.4 获得颜色表 244
- 16.4 可绘区和 pixmap 244
- 16.5 事件 245
 - 16.5.1 事件类型 245
 - 16.5.2 事件屏蔽 247
 - 16.5.3 在 Gtk+中接收 Gdk 事件 248
 - 16.5.4 鼠标按键事件 250
 - 16.5.5 键盘事件 252
 - 16.5.6 鼠标移动事件 254
 - 16.5.7 焦点变更事件 257
- 16.6 鼠标指针 257
 - 16.6.1 指针定位 257
 - 16.6.2 独占指针 258
 - 16.6.3 改变光标 259
- 16.7 字体 259
- 16.8 图形上下文 263
- 16.9 绘图 267
 - 16.9.1 画点 267
 - 16.9.2 画线 268
 - 16.9.3 矩形 268
 - 16.9.4 画弧 269
 - 16.9.5 多边形 269
 - 16.9.6 文本 270
 - 16.9.7 pixmap 像素映射图形 270
 - 16.9.8 RGB 缓冲 271

第三部分 Linux GUI 生成器 Glade

第 17 章 Glade: GUI 生成器 273

- 17.1 安装 Glade 273
 - 17.1.1 Glade 简介 273
 - 17.1.2 安装 Glade 273
 - 17.1.3 在 Gnome 主菜单下为 Glade 创建菜单项 274
 - 17.1.4 在 Gnome 面板上创建快捷按钮 275
- 17.2 用 Glade 生成图形用户接口 275
 - 17.2.1 Glade 的界面简介 275
 - 17.2.2 用 Glade 创建应用程序界面 277

第四部分 调试工具

第 18 章 程序调试 283

- 18.1 用 gdb 调试应用程序 283
 - 18.1.1 为调试程序做准备 283
 - 18.1.2 获得 gdb 帮助 284
 - 18.1.3 gdb 常用命令 284
 - 18.1.4 gdb 应用举例 286
- 18.2 用 xxgdb 调试应用程序 289

第五部分 附 录

附录 A GnomeHello 源代码 293

附录 B 在线资源 304

附录 C Gtk+/Gnome 对象总览 306

第一部分 Linux GUI 编程

框架及编程基础

第1章 Linux软件开发概述

1.1 关于Linux

Linux于1991年诞生于芬兰。大学生Linus Torvalds，由于没有足够的钱购买昂贵的商用操作系统，于是自己编写了一个小的操作系统内核，这就是Linux的前身。Linus Torvalds将操作系统的源代码在Internet上公布，受到了计算机爱好者的热烈欢迎。各种各样的计算机高手不断地为它添加新的特性，并不断地提高它的稳定性。1994年，Linux 1.0正式发布。现在，Linux已经成为一个功能强劲的32位的操作系统。

严格地说，Linux只是一个操作系统内核。比较正式的称呼是GNU操作系统，它使用Linux内核。GNU的意思是GNU's not Unix(GNU不是Unix)——一种诙谐的说法，意指GNU是一种类Unix的操作系统。GNU计划是由自由软件的创始人Stallman在20世纪80年代提出的一个庞大的项目，目的是提供一个免费的类Unix的操作系统以及在上面运行的应用程序。GNU项目在初期进展并不顺利，特别是操作系统内核方面。Linux适时而出，由于它出色的性能，使它成为GNU项目的操作系统的内核。从此以后，GNU项目进展非常迅速：全世界的计算机高手已经为它贡献了非常多的应用程序和源代码。

Linux是遵从GPL协议的软件，也就是说，只要遵从GPL协议，就可以免费得到它的软件和源代码，并对它进行自由地修改。然而，对一般用户来说，从Internet或者其他途径获得这些源代码，然后对它们进行编译和安装是技术难度很高的工作。一些应用程序的安装也都非常复杂。因而，有一些公司如Red Hat、VA等开始介入Linux的业务。它们将Linux操作系统以及一些重要的应用程序打包，并提供较方便的安装界面。同时，还提供一些有偿的商业服务如技术支持等。这些公司所提供的产品一般称为Linux的发布版本。目前比较著名的Linux发布版本有以下几种：

Red Hat——最著名的Linux服务提供商，Intel、Dell等大公司都对其有较大投资，该公司前不久收购了开放源代码工具供应商Cygnus公司。

SlackWare——历史比较悠久，有一定的用户基础。

SUSE——在欧洲知名度较大。

TurboLinux——在亚洲，特别是日本用户较多。该公司在中国推出了TurboLinux 4.0、4.02和6.0的中文版，汉化做得很出色。

Debain——完全由计算机爱好者和Linux社区的计算机高手维护的Linux发布版本。

Linux进入中国后，在我国计算机界引起了强烈的反响，最近两年，也出现了许多汉化的Linux发布版本，影响较大的有以下几种：

XteamLinux——北京冲浪平台公司推出的产品，中国第一套汉化的Linux发布版本。

BluePoint——1999年底正式推出的产品，内核汉化技术颇受瞩目。

红旗Linux——中国科学院软件研究所和北大方正推出的 Linux 发布版本。

从本质上来说，上面所有发布版本使用的都是同样的内核(或者版本略有不同)，因而，它们在使用上基本上没有什么区别。但它们的安装界面不一样，所包含的应用程序也有所不同。

Linux之所以大受欢迎，不仅仅因为它是免费的，而且还有以下原因：

1) Linux是一个真正的抢占式多任务、多线程、多用户的操作系统。

2) Linux性能非常稳定，功能强劲，可以与最新的商用操作系统媲美。

3) Linux有非常广泛的平台适应性。它在基于 Intel公司的x86(也包括AMD、Cyrix、IDT)的计算机、基于Alpha的计算机，以及苹果、Sun、SGI等公司的计算机上都有相应的发布版本，甚至在AS/400这样的机器上都能找到相应的版本。Linux还可以在许多PDA和掌上电脑以及嵌入式设备上运行。

4) 已有非常多的应用程序可以在Linux上运行，大多数为SCO Unix开发的应用程序都能在Linux上运行(借助于iBCS软件包)，甚至还比在SCO Unix上运行速度更快。借助Dosemu，可以运行许多DOS应用程序，而借助Wabi或Wine，还可以运行许多为Windows设计的软件。

5) Linux是公开源代码的，也就是说，不用担心某公司会在系统中留下后门(软件开发商或程序员预留的，可以绕开正常安全机制进入系统的入口)。

6) 只要遵从GPL协议，就可以自由地对Linux进行修改和剪裁。

当然，Linux的优点决不止于此。对计算机专业人员来说，Linux及其相关应用程序也是学习编程的绝好材料，因为这些软件都提供了完整的源代码。

Linux的出现为我国软件产业赶超世界先进水平提供了极好的机遇，也为我国软件产业反对微软的垄断提供了有力的武器。

1.2 关于Linux的桌面环境

目前使用Linux主要在于服务器端。在Internet上有很多服务器都在使用Linux。但是，一个操作系统要想得到普及，并占据一定的市场份额，必须要使非计算机专业人士都可以轻松掌握这种系统。而Linux作为一种类Unix操作系统，对它的操作一般都是通过复杂的Shell命令进行的。因而，应该有一种简便易学的图形用户接口(Graphics User Interface, GUI)，使用户使用鼠标就可以完成大多数工作。

在Linux中，GUI由以下几个部分组成：

- 窗口系统—组织显示屏上的图形输出并执行基本的文本和绘图功能。
- 窗口管理器—负责对窗口的操作(比如最小化、最大化、关闭按钮的形状，窗口边框外观等)以及输入焦点的管理。
- 工具包—带有明确定义的编程界面的常规库。
- 风格—指定应用程序的用户界面外观和行为。

在Linux发展的初期，众多的计算机专家为它贡献了多种图形用户接口，如FVWM95、AfterStep等。这些接口模仿了Windows 95、Macintosh、NestStep、Amiga、Unix CDE等桌面环境。这些GUI在一定程度上来说只是其他图形接口的仿制品，不能提供优秀的操作系统所需要的特性。其后，自由软件社区的一批计算机专家开始了KDE项目(K Desktop Environment, K桌面环境)，目的是提供一个开放源代码的图形用户接口和开发环境。该项目取得了极大的

成功，KDE成为许多Linux发布版本的首选桌面环境。GNU/Linux项目因此而得到蓬勃发展。但是，KDE是基于Troll Technologies公司的Qt库的。Qt库是一个跨平台的C++类库，可以用于多种Unix、Linux、Win32等操作系统。Qt并不是遵从GPL或LGPL协议的软件包。它的许可条件是：如果使用它的免费版本开发应用程序或程序库，则所开发的软件必须开放源代码；如果使用它的商用版本，则可以用以开发私有的商用软件。另外，Qt库是属于Troll公司的产品，一旦Troll公司破产，或者被收购，自由软件事业将受到严重打击。

1997年由墨西哥国立自治大学的Miguel de Icaza领导的项目组开始了Gnome开发计划。Gnome是GNU Network Object Model Environment(GNU，网络对象模型环境)的缩写。该计划的最初目的是创建一种基于应用程序对象的架构，类似于微软公司的OLE和COM技术。然而，随着项目的进展，项目的范围也迅速地扩大；项目开发过程中有数百名程序员加入进来，编写了成千上万行的源代码。该项目进展很快，1998年发布了Gnome 1.0。目前的最新版本是于1999年10月发布的October Gnome。现在，Gnome已成为一个强劲的GUI应用程序开发框架，并且可以在任何一种Unix系统下运行。Gnome使用的图形库是Gtk+——最初为了编写GIMP而创建的一套构件库，它是基于LGPL创建的，可以用它来开发开放源代码的自由软件，也可以开发不开放源代码的商用软件。Gnome的界面与KDE的界面是类似的(Gnome的目的之一就是创建一套类似KDE的桌面环境)，熟悉KDE的用户无需学习就能够使用Gnome。由于以上几个原因，Gnome已经成为大多数Linux发布版本的首选桌面环境。

由于Gnome项目的成功，1998年11月Qt库的开发者Troll公司宣布修改许可证协议，Qt库将成为自由软件。但是获取Qt库的许可证很不方便，况且Gnome的进展也很不错，因而，只要有可能，应该避免使用Qt库以及KDE。

从用户的角度看，Gnome是一个集成桌面环境和应用程序的套件。从程序员的角度看，它是一个应用程序开发框架(由数目众多的实用函数库组成)。即使用户不运行Gnome桌面环境，用Gnome编写的应用程序也可以正常运行，但是这些应用程序是可以很好地和Gnome桌面环境集成的。Gnome桌面环境包含文件管理器，它用于任务切换、启动程序以及放置其他程序的“面板”、“控制中心”(包括配置系统的程序以及一些小东西)等。这些程序在易用的图形界面背后隐藏了传统的UNIX Shell。Gnome的开发结构使开发一致的、易用的和可互相操作的应用程序成为可能。

1.3 Linux系统中的软件开发

1.3.1 开发所使用的库

在Linux下开发GUI程序的首要问题是采用什么样的图形库。在Linux的发展历史中曾经出现过多种图形库，但是由于自由软件的特点(没有技术方面的承诺)，使得无人继续对它们进行维护，或者其他方面的原因，这些库都已慢慢地被人遗忘了。

Gtk+(GIMP ToolKit，GIMP工具包)是一个用于创造图形用户接口的图形库。Gtk+是基于LGPL授权的，因此可以用Gtk+开发开放源码软件、自由软件，甚至商业的、非自由的软件，并且不需要为授权费或版权费花费一分钱。之所以被称为GIMP工具包因为它最初用于开发“通用图片处理程序”(General Image Manipulation Program，GIMP)，但是Gtk+已在大量软件项目，包括Gnome中得到了广泛应用。Gtk+是在Gdk(GIMP Drawing Kit，GIMP绘图包)的基

础上创建的。Gdk是对低级窗口函数的包装(对X window系统来说就是Xlib)。

读者可能会看到,在本书中既有GTK,又出现了Gtk+。一般用GTK代表软件包和共享库,用Gtk+代表GTK的图形构件集。

GTK的主要作者是:

```
Peter Mattis pe@xsf.berkeley.edu  
Spencer Kimball spend@xsf.berkeley.edu  
Josh MacDonald jma@xsf.berkeley.edu
```

Gtk+图形库使用一系列称为“构件”的对象来创建应用程序的图形用户接口。它提供了窗口、标签、命令按钮、开关按钮、检查按钮、无线按钮、框架、列表框、组合框、树、列表视图、笔记本、状态条等构件。可以用它们来构造非常丰富的用户界面。

在用Gtk+开发Gnome的过程中,由于实际需要,在上面的构件基础上,又开发了一些新构件。一般把这些构件称为Gnome构件(与Gtk+构件相对应)。这些构件都是Gtk+构件库的补充,它们提供了许多Gtk+构件没有的功能。从本质上来说,Gtk+构件和Gnome构件是完全类似的东西。

GTK本质上是面向对象的应用程序编程接口(API)。虽然完全是用C写成的,但它仍然是用类和回调函数(指向函数的指针)的方法实现的。

1.3.2 Gnome的开发结构

只使用Gtk+构件也可以开发出优秀的Linux应用程序,但是Gnome构件,特别是GnomeApp、GnomeUIInfo等,使开发界面一致的应用程序变得更加容易。Gnome的一些新特性,如popt参数分析,保存应用程序设置等也是Gtk+构件所没有的。

Gnome的应用程序开发结构核心是一套库,都是由通用的ANSI C语言编写的,并且倾向于使用在类UNIX的系统上。其中涉及图形的库依赖于XWindow系统。Gnome差不多对任何语言都提供了Gnome API接口,其中包括Ada、Scheme、Python、Perl、Tom、Eiffel、Dylan以及Objective C等。至少有三种不同的C++封装。本书只介绍有关库的C语言接口,不过,对使用其他语言绑定的用户来说,它也很有用,因为从C到其他语言之间的转换都是非常直接的。本书包含Gnome库1.0版本(包括兼容的bug补丁版,比如1.0.9——所有1.0.x版本都是兼容的)。

Gnome的开发架构包含以下一些内容:

1. 非Gnome 库

Gnome并不是从头开始的,它充分继承了自由软件的传统——其中许多内容来自于Gnome项目开始之前的一些函数库。其中一些库Gnome应用程序开发架构的一部分,但是不属于Gnome库——我们称之为非Gnome库。可以在Gnome环境中使用这些库函数。主要有以下几种:

Glib Glib是Gnome的基础,它是一个C工具库,提供了创建和操作常用数据结构的实用函数。它也涉及到了可移植性问题,例如,许多系统缺乏snprintf()函数,但是glib包含了一个,称为g_snprintf(),它能保证在所有平台上使用,并且比snprintf()更安全(它总是将目标字符串以NULL结尾)。Gnome 1.0中使用glib的1.2版本,可以和任何1.2系列的glib一起工作(1.2.1、1.2.2,等等)。

Gtk+ Gtk+(GIMP Toolkit的缩写),是在Gnome应用程序中使用的GUI工具包。Gtk+最初是为了设计GIMP而引入的(GNU 图片处理程序),但是现在已变成通用的库。Gtk+依赖于glib。Gtk+包中包含了Gdk,它是对底层的 X Window系统库Xlib的简化。由于Gtk+使用了Gdk而不是直接调用Xlib,因此Gdk的移植版本允许Gtk+运行在不同于X 但只有相对较少的修改的窗口系统上。Gtk+和Gimp已经移植到了Win32平台(32位的Windows平台,包括 Windows 95/98、Windows NT/2000)上。

对Gnome应用程序来说, Gtk+具有以下特性:

- 1) 动态类型系统。
- 2) 用C语言编写的对象系统,可实现继承、类型检验,以及信号 /回调函数的基础结构。
- 3) 类型和对象系统不是特别针对 GUI的。
- 4) GtkWidget对象使用对象系统,它定义了Gtk+的图形组件的使用接口。
- 5) 大量的GtkWidget子类(构件)。

Gnome在基本Gtk+构件集合的基础上添加了许多其他构件。 Gnome 1.0是在Gtk+ 1.2版本的基础上完成的。

ORBit ORBit是一个用C开发的CORBA 2.2 ORB。和其他ORB相比,它短小精悍,但速度更快,同时还支持 C语言映射。ORBit是以一整套库函数的方式实现的。CORBA,或称作通用对象请求中介构架(Common Object Request Broker Architecture),是一套对象请求中介,或称为ORB的规范。一个ORB更类似于动态链接程序,但是它以对象的方式操作,而非子程序调用。在执行过程中,程序能够请求一个特定的对象服务; ORB可定位对象并且创建对象和程序连接。例如,一个电子邮件程序可以请求 addressbook对象,并且利用它查找人名。与动态链接库不同, CORBA可以在网络内很好地运行,并且允许不同编程语言和操作系统之间进行交互。如果熟悉Windows操作系统下的DCOM,那么CORBA与之类似。

Imlib Imlib(图片库)提供一些例程,其中包括加载、存储、显示,以及定绘制各种流行的图像格式(包括 GIF、JPEG、PNG以及TIFF)的函数。它包括两种版本: Xlib-only版本和基于Gdk的版本。Gnome使用 Gdk版本。

2. Gnome库

下面所介绍的库是Gnome-libs包的一部分,并且是专门为Gnome项目开发的。

libgnome libgnome是一些与图形用户接口无关的函数集合, Gnome应用程序可以调用其中的函数。它包含分析配置文件的代码,也包含与一些外部实用程序的接口,比如国际化编程接口(通过GNU gettext包)、变量解析(通过popt包)、声音编程接口(通过Enlightenment Daemon, esound)等。Gnome-libs包考虑了与外部库之间的交互,因此程序员无需关心库的实现或可用性。

libgnomeui libgnomeui包含了与GUI相关的Gnome代码。它由为增强和扩展Gtk+功能而设计的构件组成。Gnome构件通常使用用户接口策略,以提供更方便的 API函数(这样程序员需要指定的东西较少)。当然,这也让应用程序界面更一致。

libgnomeui主要包含:

1) GnomeApp构件 一般用来为应用程序创建主窗口。它使用 GnomeDock构件,允许用户重新排列工具栏,还可以将工具条从窗口上拖开。

2) GnomeCanvas构件 用来编写复杂的、无闪烁的定制构件。

3) Gnome 内置的 pixmap(包括打开、关闭、保存以及其他操作的图标) 用于创建和使用对话框的例程。GnomePixmap 构件比 GtkPixmap 功能更多。

libgnomeui 中还有几种其他构件, 如 GnomeEntry、GnomeFilePicker 等。这些构件都比 Gtk+ 构件库中的构件功能更强, 也更方便。

libgnorba libgnorba 提供与 CORBA 相关的实用程序, 包括安全机制和对象激活。对象激活是指获得实现给定接口对象的引用过程, 它包括执行服务器程序, 加载共享库模块, 或为已有程序请求新的对象实例等。

libzvt 这个库包含一个终端构件 (ZvtTerm), 可以在 Gnome 程序中使用它。

libart_lgpl 这个库包含由 Raph Levien 编写的图形绘制例程。在这里包含的是在 LGPL 许可下发布的, 用在 GnomeCanvas 构件中的, Raph Levien 也销售它的增强版本。实质上它是一个矢量图形光栅图形库, 功能类似于 PostScript 语言。

3. 其他库

这些库一般使用在 Gnome 应用程序中, 但它不是 Gnome-libs 专属的部分。

Gnome-print Gnome-print 目前还是实验性的, 但是非常有前途。它使用 libart_lgpl 库, 可以和 GnomeCanvas 一起工作得很好。它提供一个虚拟输出设备 (称 “打印上下文”), 因此一段代码能输出到一个打印预览构件或 PostScript 文件, 还可以输出到其他打印机格式。Gnome-print 也包含与打印相关的 GUI 元素, 例如打印设置对话框、虚拟字体接口 (处理 X 字体不可打印的问题)。

Gnome-xml Gnome-xml 是还未经验证的 XML 引擎, 它由 WWW 协会的 Daniel Veillard 编写。它能按照树状结构分析 XML, 也能按照 XML 输出树状结构。它对任何需要加载和保存结构化数据的应用程序来说是很很有用的, 许多 Gnome 应用程序把它作为文件格式使用。这个库不依赖于任何其他的库 (甚至 glib), 所以它只是在名义上是一个 Gnome 库。然而, 可以认为大多数 Gnome 用户都安装了它, 因此如果应用程序使用了这个库, 对用户来说也没有什么不方便。

Guile Guile 是 Scheme 编程语言在一个库中的实现, 它使任何应用程序都能带有一个嵌入式的 Scheme 解释器。它是 GNU 项目的正式扩展语言, 并且有一些 Gnome 应用程序也使用它。为应用程序添加扩展语言听起来挺复杂, 但是有了 Guile 后就微不足道了。一些 Gnome 应用程序也支持 Perl 和 Python, 一旦实现了应用程序, 同时支持几种语言就会变得很容易。Guile 在 Gnome 开发者心目中有着特殊的地位。

Bonobo Bonobo 是一种对象嵌入式结构, 类似于 Microsoft 的 OLE。例如, 它允许你在电子表格中嵌入图表。它将在 Gnome 中普遍使用。任何应用程序将能通过适当的 Bonobo 组件调用 Gnome 库, 显示 MIME 类型数据, 例如纯文本、HTML 或图像。

1.4 开发 Linux 应用程序的编程语言和编程工具

Linux 是一种类 Unix 的操作系统。传统 Unix 下的开发语言是 C 语言。因为 C 语言是平台适用性最强的语言, 差不多每种平台上都会有一个 C 编译器。C 语言也更易移植, 因而, 在 Linux 下编程的最佳语言应该是 C 语言, Linux 上的很多应用程序就是用 C 语言写的。当然, 也可以使用其他语言。

因为 Gtk+ 和 Gnome 是用 C 语言编写的, 所以在开发 Linux 下的 GUI 程序时使用 C 语言是非常

方便的。但是 Gtk+ 也提供与许多其他语言的接口，如 Ada、Scheme、Python、Perl、Tom、Eiffel、Dylan 以及 Objective C 等。如果用 C++ 语言开发基于 Gtk+ 应用程序，可以使用一个名为 Gtk-- 的函数库，它是 GTK 工具包的 C++ 风格的封装。如果要用 Gtk+ 库和其他语言，最好参考相应的文档。本书只介绍使用 C 语言开发 Linux 程序。

一般的 Linux 发布版本中都提供了 C 编译器 gcc 或 egcs。使用 gcc 或 egcs 可以编译 C 和 C++ 源代码，编译出的目标代码质量非常好，编译速度也很快。

各种 C 编译器都要使用一些 C 语言实用函数。为了保证程序的可移植性，gcc 没有使用通用的 C 函数库，而是使用一种称为 glib 的函数库。glib 也是 Gtk+ 的基础。它包含一些标准函数的替代函数（如字符串处理函数）和基本数据结构的实现（单向链表、双向链表、树、哈希表等）。glib 中所包含的函数消除了某些函数的安全漏洞，使其更加可靠，在不同平台上移植也更加方便。

还有许多使用工具可以提高 Linux 下的编程效率，如 gdb 是优秀的 C 语言调试器，有丰富的调试指令；automake 和 autoconf 用于由源代码结构配置编译选项，生成编译所需的 Makefile 文件。

到目前为止，还没有像 Windows 平台上的 Visual Basic、Delphi 等一样的可视化的快速应用程序开发工具。开发 Linux 应用需要用文本编辑器书写源代码，然后再用编译器生成应用程序。眼下有一个开发小组正致力于开发一个 Linux 下的类似于 Visual Basic 的开发工具——gBasic，另外，预计 Inprise 公司（即 Borland）的 Delphi for Linux 也即将面市。

有几种正在开发的 RAD（Rapid Application Development）工具，其中最希望的是 Glade——一种 GUI 生成器，可以快速生成创建界面的 C 源程序。

1.5 本书的结构

本书包含了以下内容：

- Gnome 应用程序开发的框架。
- 开发 Linux 应用程序的方法和步骤。
- glib，Gtk+/Gnome 构件的使用方法。
- GDK 基础知识。
- Linux 常用编程工具：调试工具 gdb，GUI 生成器 Glade。

本书附录包含 Gtk+ 和 Gnome 对象介绍（每个对象有一个简短描述），还有一些在线编程资源。

本书提供了大量可供参考的源代码。如果觉得内容不够充分，请参考相应的头文件。实际上，Gtk+/Gnome 和 glib 的头文件都是非常简单易懂的，从函数名称就可以猜到它的用处和用法。如果还觉得不够，可以钻研它们的源代码，这对了解它们的实现方法也是极有好处的。

第2章 Gtk+/Gnome开发简介

2.1 安装Gtk+/Gnome库

要想用Gtk+/Gnome编程，首先要保证系统中已经安装了Gtk+和Gnome库。

一般情况下，Linux发布版本的光盘中都应该包含了所需要的库的源代码。例如 Red Hat Linux 6.0/6.1和TurboLinux 4.0中都有Gtk+和Gnome的最新版本。在安装系统的过程中，当提示安装类型时，一般有“服务器”、“工作站”、“开发工作站”、“自定义”和“完全安装”等几个选项。选择“开发工作站”或“完全安装”，完全安装大多数用于软件开发的库、头文件和实用程序，如Gtk+库、Gnome库、automake、autoconfig、gcc编译器、gdb调试器等。

如果觉得系统中已有的库的版本已经过时，可以从Internet上下载最新版本，然后安装。可以从Gtk+的Web站点<http://www.gtk.org>下载最新版本的Gtk+。Gtk+所使用的版本号与Linux的版本编号方法类似，偶数版本号（如1.0和1.2）表示稳定的版本，而奇数版本号（如0.9和1.1）表示正在开发的版本。有时还增加一个附加数字表示对这一版本进行了修正，如1.2.1。当前Gtk+的最新版本是1.2.3。

从网上下载的文件名一般是gtk+1.2.3.tar.gz或者其他类似形式，文件名中包含了该软件包的名称和版本号信息。因为它的后缀是.tar.gz，所以它是一个归档的压缩文件。用gunzip命令对它解压缩：

```
gunzip gtk+1.2.3.tar.gz
```

将会产生一个解压缩的以.tar结尾的归档文件。用tar命令将它扩展为它的目录结构：

```
tar -xvf gtk+1.2.3.tar
```

这个命令建立了建库所需要的目录结构。进入上面建立的目录，执行configure脚本生成编译所需的makefile：

```
./configure
```

下面可以建立库了。输入make命令：

```
make
```

建库后，需要安装刚才建立的库。输入以下命令：

```
make install
```

然后，还需要运行/sbin/ldconfig以使Gtk+能正常工作。

当然，完成上面工作的前提是系统上已经安装了glib。不过，一般情况下都可以保证做到这一点。如果需要安装新版本的glib库，操作步骤和安装Gtk+库一样。

在Gtk+源代码目录下的examples子目录下，有很多Gtk+构件示例。这些代码都有很详细的注释，通读这些代码对学习Gtk+编程也是很有帮助的。本书中关于Gtk+构件的演示代码都来自这些示例。

Gnome的最新版本可以从<http://www.gnome.org>下载。取得新版本软件后，解压缩和安装的方法与Gtk+类似。

2.2 第一个Gtk+应用程序

2.2.1 一个什么也不能做的窗口

用Gtk+库编程和同时使用Gtk+/Gnome库编程的方法完全相同，只是细节上略有不同。我们将从一个最简单的程序开始介绍GTK，然后用一个简单的例子介绍Gnome库编程的方法。

本程序将创建一个200×200像素的窗口，除了用shell命令kill以外，没有其他的退出程序的方法。

```
/* 例子开始 base.c */
#include <gtk/gtk.h>
int main( int argc, char *argv[] )
{
    GtkWidget *window;
    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_show (window);
    gtk_main ();
    return(0);}
/*示例结束 */
```

上面的源代码中的/*和*/之间的内容都是注释。Linux中常用的C编译器能够识别/* */和//两种格式的注释。

可以用gcc编译上述程序：

```
gcc base.c -o base `gtk-config --cflags --libs`
```

注意，gtk-config --cflags--libs选项左右的符号不是单引号，而是反引号（键盘上“1”键左边的键）。编译选项的含义选项将在下面编译“Hello World”时解释。

编译结束后，输入以下命令来执行上面创建的应用程序（结果如图2-1所示）：

```
./base
```

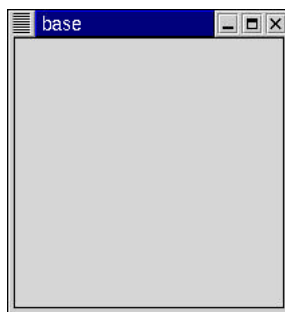


图2-1 第一个Gtk应用程序，它什么也不能做

2.2.2 示例代码的含义

在程序的源代码中，所有用到的函数和数据类型以及结构等都应该声明。也就是说，如果使用了一个按钮构件，应该包含按钮所对应的gtkbutton.h头文件。如果在程序中使用了多种构件和函数库，包含这些头文件将会是一件很麻烦的事，也很容易出错。有一个特殊的头文件，gtk/gtk.h，其中包含了Gtk+编程中所有需要的头文件，也包含gdk.h和glib.h等。在源文件的前面包含这个文件就可以了。后面会看到，如果要用到Gnome的构件和库函数，包含gnome.h就可以了。在gnome.h中已经包含了gtk.h文件和其他相关头文件。

第一行#include <gtk/gtk.h> 包含了所有需要的头文件。

下一行：

```
gtk_init (&argc, &argv);
```

先调用函数gtk_init(gint *argc, gchar ***argv)。所有GTK应用程序都要调用该函数。它为我们设置一些缺省值例如视觉和颜色映射，然后调用 gdk_init(gint *argc, gchar ***argv)。这

个函数将函数库初始化,设置缺省的信号处理函数,并检查通过命令行传递给应用程序的参数。

主要检查下面所列的一些值:

```
--gtk-module
--g-fatal-warnings
--gtk-debug
--gtk-no-debug
--gdk-debug
--gdk-no-debug
--display
--sync
--no-xshm
--name
--class
```

它从变量表中删除以上参数,并分离出所有不识别的值,应用程序可以分析或忽略这些值,由此生成一些Gtk应用程序可以接受的标准参数。

下面两行代码将创建和显示一个窗口:

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_show (window);
```

GTK_WINDOW_TOPLEVEL参数指明让窗口使用“窗口管理程序”指定的状态设置和位置布置。没有子窗口的窗口缺省设置为 200×200 像素大小,而不是 0×0 大小。gtk_widget_show()函数让Gtk知道那我们已经设置该构件(窗口)的属性,现在可以显示它了。

最后一行代码进入Gtk主处理循环:

```
gtk_main ();
```

gtk_main()是在每个Gtk应用程序都要调用的函数。当程序运行到这里时,Gtk将进入等待状态,等候X事件(比如点击按钮或按下键盘的某个按键)、Timeout或文件输入/输出发生。

在这个简单例子里,所有事件都被忽略。用鼠标点击窗口右上角的“×”按钮也不能将窗口关闭。唯一关闭它的办法就是使用Shell命令kill删除它的进程。

这里顺便指出:在代码中可以使用C语言风格的注释(/* */)和C++语言风格的注释(//)。不过,最好能够使用C语言风格注释,否则可能在某些平台上编译时会出现错误。

2.2.3 GTK的Hello World

上面的例子实在是太简陋了,它甚至什么也不能做。下面我们创建一个Gtk版本的“Hello World”以进一步说明Gtk+的编程方法。在这个例子中,我们在窗口里面放一个按钮构件。下面是示例的源代码。

1. 源代码

```
/*示例开始 helloworld helloworld.c */
#include <gtk/gtk.h>
/*回调函数在本例中忽略了传递给程序的所有参数。下面是回调函数 */
void hello( GtkWidget *widget, gpointer data )
{
    g_print ("Hello World\n");
}
```

```

}
gint delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
/*如果在"delete_event"处理程序中返回FALSE, GTK 将引发一个"destroy"
*信号, 返回TRUE意味着你不想关闭窗口。
* 这些在弹出"你真的要退出?"对话框时很有作用 */
g_print ( "delete event occurred\n");
/* 将TRUE改为FALSE, 主窗口就会用一个"delete_event"信号, 然后退出 */
return(TRUE);
}
/* 另一个回调函数 */
void destroy( GtkWidget *widget, gpointer data )
{
gtk_main_quit();
}
int main( int argc, char *argv[] )
{
/* GtkWidget是构件的存储类型 */
GtkWidget *window;
GtkWidget *button;
/* 在所有的Gtk应用程序中都应该调用。它的作用是解析由命令行传递
* 进来的参数并将它返回给应用程序 */
gtk_init(&argc, &argv);
/* 创建一个主窗口 */
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
/* 当给窗口一个"delete_event"信号时(这个信号是由窗口管
* 理器发出的, 通常是在点击窗口标题条右边的 "x" 按钮, 或
* 者在标题条的快捷菜单上选择 "close"选项时发出的), 我们
* 要求调用上面定义的delete_event()函数传递给这个回调函数
* 的数据是NULL, 回调函数会忽略这个参数 */
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                    GTK_SIGNAL_FUNC (delete_event), NULL);
/* 这里, 我们给 "destroy"事件连接一个信号处理函数,
* 当我们在窗口上调用gtk_widget_destroy()函数
* 或者在 "delete_event"事件的回调函数中返回 FALSE
* 时会发生这个事件 */
gtk_signal_connect (GTK_OBJECT (window), "destroy",
                    GTK_SIGNAL_FUNC (destroy), NULL);
/* 设置窗口的边框宽度 */
gtk_container_set_border_width (GTK_CONTAINER (window), 10);
/* 创建一个标题为 "Hello World"的按钮 */
button = gtk_button_new_with_label ("Hello World");
/* 当按钮接收到 "clicked"时, 它会调用hello()函数,
* 传递的参数为NULL。函数hello()是在上面定义的 */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (hello), NULL);
/* 当点击按钮时, 通过调用gtk_widget_destroy(window)函数销毁窗口。
* 另外, "destroy"信号可以从这里发出, 也可以来自于窗口管理器 */
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),
                           GTK_OBJECT (window));

```

```

/* 将按钮组装到窗口中 (一个gtk容器中) */
gtk_container_add (GTK_CONTAINER (window), button);
/* 最后一步就是显示新创建的构件 */
gtk_widget_show (button);
/* 显示窗口 */
gtk_widget_show (window);
/* 所有的GTK应用程序都应该有一个gtk_main()函数。
* 程序的控制权停在这里并等着事件的发生 (比如一次按键或鼠标事件) */
gtk_main ();
return(0); }
/* 示例结束 */

```

2. 编译 “Hello World” 应用程序

用以下语法编译：

```
gcc -Wall -g helloworld.c -o helloworld `gtk-config --cflags` \
`gtk-config --libs`
```

上面使用了Gtk自带的gtk-config程序。这个程序“知道”用Gtk编译应用程序时需要使用什么编译程序开关选项。

gtk-config --cflags将输出一个包含编译器需要的查找路径的列表，并且 gtk-config --libs将输出编译程序要链接的库的列表和找到这些库的路径。在上面的例子中，这些选项可以写在一句话里，比如`gtk-config --cflags --libs`。

编译中通常要链接的库包含下面这些：

GTK库 (-lgtk)：构件库,建立在Gdk的基础上。

GDK库 (-lgdk)：包装的Xlib库。

gmodule库 (-lgmodule)：用于在加载运行时扩展函数。

glib库 (-lglib)：包含了各种实用函数。在这里仅仅使用了 g_print()。Gtk是在Glib的基础上创建的，所以需要使用glib库。

Xlib库 (-lX11)：GDK库要调用该库。

Xext库 (-lXext)：包含为共享内存中的pixmap图片和其他X扩展的代码。

数学库 (-lm)：被GTK调用。

2.2.4 Gtk+的信号和回调函数原理

在详述helloworld的细节之前，我们先讨论信号和回调函数。GTK是一种事件驱动工具包，这意味着它将在gtk_main函数中一直等待，直到事件发生和控制权被传递给相应的函数。

1. 信号 (signal)

控制权的传递是使用“信号”的方法。一旦事件发生，比如鼠标器按钮被按下，被按下的构件（按钮）将引发适当的信号。这是GTK实现其绝大多数工作的方法。

有一些信号是大多数构件都具备的，比如 destroy，还有一些是某些构件专有的，比如在开关按钮（togglebutton）的toggled信号。

要让一个按钮执行一个操作，我们需要建立一段信号处理程序，以捕获它的信号，然后调用相应的函数。这由类似以下所示的函数实现：

```

gint gtk_signal_connect( GtkObject *object, gchar *name,
                        GtkSignalFunc func, gpointer func_data );

```

第一个参数 *object 是将要发出信号的构件，第二个参数 *name 是希望捕获的信号的名称，第三个参数 func 是捕获信号时要调用的函数，第四个参数 func_data 是要传递给函数的用户数据参数。

在第三个参数里指定的函数称为“回调函数”，它的形式通常是：

```
void callback_func( GtkWidget *widget, gpointer callback_data );
```

这个函数的第一个参数是一个指向发出信号的构件的指针，第二个参数是一个指向传递给回调函数的用户数据的指针。

注意 上述对“信号”的回调函数的声明仅仅是一个通用的规则，因为一些构件的特殊“信号”产生不同调用参数。例如，Clist构件的select_row构件同时提供行和列参数。

另一个在 helloworld 例子里使用的函数调用是：

```
gint gtk_signal_connect_object( GObject *object,
                                gchar *name,
                                GtkSignalFunc func,
                                GObject *slot_object );
```

gtk_signal_connect_object() 跟 gtk_signal_connect() 一样，除了回调函数仅仅使用一个参数：指向GTK对象的指针。

这样，当用这个函数连接到一个信号时，回调函数将以下面的形式出现：

```
void callback_func( GObject *object );
```

在这里，参数 object 通常是一个构件。

不过，我们通常不为 gtk_signal_connect_object() 设置回调函数。它们通常用于调用接受信号或对象作为参数的 Gtk 函数，就像在 helloworld 里的那样。有两种函数能连接到信号的目的仅仅是允许回调函数可以有不同数量的参数。

因为在 GTK 库里许多函数仅仅接受单个 GtkWidget 指针作为参数，所以可能想用 gtk_signal_connect_object() 连接到某些回调函数。在这种情况下，需要提供附加的数据传递到回调函数。

2. 事件

除了上面描述的信号机制之外，还有一套与 X 事件（X Window 中发生的动作，比如鼠标某个按键按下或弹起，鼠标移动等）机制相对应的事件。回调函数也可以与这些事件连接起来。

这些事件是：

button_press_event	button_release_event
motion_notify_event	delete_event
destroy_event	expose_event
key_press_event	key_release_event
enter_notify_event	leave_notify_event
configure_event	focus_in_event
focus_out_event	map_event unmap

要将回调函数连接到上面的某一个事件，需要使用 gtk_signal_connect 函数，并使用上面的事件名称作为命名参数。事件的回调函数与信号的回调函数在形式上略有不同：

```
Void func(
    GtkWidget *widget, GdkEvent *event,
    gpointer callback_data );
```

GdkEvent是 C中的联合体结构，其类型依赖于发生的事件是哪一个事件。要想知道哪一个事件已经引发，可以查看类型参数，因为每个可能的可选事件都有一个反映引发事件的类型参数。

事件结构的另一个组件依赖于事件类型。事件类型的可能取值有：

GDK_NOHING	GDK_DELETE	GDK_DESTROY
GDK_EXPOSE	GDK_MOTION_NOTIFY	GDK_BUTTON_PRESS
GDK_2BUTTON_PRESS	GDK_3BUTTON_PRESS	GDK_BUTTON_RELEASE
GDK_KEY_PRESS	GDK_KEY_RELEASE	GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY	GDK_FOCUS_CHANGE	GDK_CONFIGURE
GDK_MAP	GDK_UNMAP	GDK_PROPERTY_NOTIFY
GDK_SELECTION_CLEAR	GDK_SELECTION_REQUEST	GDK_SELECTION_NOTIFY
GDK_PROXIMITY_IN	GDK_PROXIMITY_OUT	GDK_DRAG_BEGIN
GDK_DRAG_REQUEST	GDK_DROP_ENTER	GDK_DROP_LEAVE
GDK_DROP_DATA_AVAIL	GDK_CLIENT_EVENT	GDK_VISIBILITY_NOTIFY
GDK_NO_EXPOSE		
GDK_OTHER_EVENT	/* 最好不要使用它 */	

因此，要将回调函数与一个事件连接起来，需要使用以下形式的函数：

```
gtk_signal_connect( GTK_OBJECT(button),
    "button_press_event",
    GTK_SIGNAL_FUNC(button_press_callback),NULL);
```

这里假定button是一个按钮构件。现在，当鼠标移动到按钮上方，鼠标按钮按下时，将调用button_press_callback函数。

这个回调函数可以作如下声明：

```
static gint button_press_callback( GtkWidget *widget,
    GdkEventButton *event, gpointer data);
```

注意，我们可以将第二个参数声明为 GdkEventButton类型，因为对被调用的函数来说，它已经知道将发生什么类型的事件。

函数返回的值指示事件是否由 GTK的事件处理机制做进一步的传播。

返回TRUE指示事件已被处理，并且不会进一步传播。

返回FALSE将继续正常的事件处理。

2.2.5 Hello World代码解释

我们已解释了 Gtk+编程中的事件、信号以及回调函数的机制，下面我们将详细解释helloworld程序。

下面是当clicked按钮时调用的回调函数。在这个例子里，我们忽略了构件和参数，但是实际处理它们并不难。在下一个例子中应用数据参数将告诉我们按下了哪一个按钮。

```
void hello( GtkWidget *widget, gpointer data )
```



```
{
g_print ("Hello World\n");
}
```

下一个回调函数有一点特殊。当窗口管理程序将事件传递给应用程序时，`delete_event`事件发生。在这里，我们有机会选择对这些事件做些什么动作。我们可以忽略它们，也可以做出一些响应，或简单地退出应用程序。

回调函数返回的值让 GTK 知道发生了什么动作。返回 `TRUE`，意味着不想引发 `destory` 信号，让应用程序继续运行。返回 `FALSE`，要求引发 `destory` 信号，然后调用 `destory` 的信号处理函数。

```
gint delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{ g_print ("destory occurred\n");
return(TRUE);
}
```

下面是另一个回调函数，通过调用 `gtk_main_quit()` 使应用程序退出。这个函数告诉 GTK 当控制被交回时将从 `gtk_main` 退出。

```
void destory( GtkWidget *widget, gpointer data )
{ gtk_main_quit ();
}
```

`main()` 是应用程序的入口。与其他任何 C 语言程序一样，该程序应该有一个 `main()` 函数作为入口。

```
int main( int argc, char *argv[] ) {
```

下一步声明了两个指向 `GtkWidget` 类型结构的指针。它们用于创建一个窗口和一个按钮：

```
GtkWidget *window;
GtkWidget *button;
```

下面又是 `gtk_init` 函数。和以前介绍的一样，这个函数初始化 GTK，并且分析在命令行中传递进来的参数。命令行中传递过来的任何参数，只要是它能识别的，都会从列表中删除，并且修改 `argc` 和 `argv` 的值，就像这些参数从不存在一样，然后应用程序分析剩余的参数。

```
gtk_init (&argc, &argv);
```

下面的语句创建新窗口。这是相当直接了当的，因为 `GtkWidget *window` 结构分配内存，现在它指向了一个有效的结构类型。到这里为止，已经创建了一个新窗口，但是直到在程序结束前调用 `gtk_widget_show(window)`，它才会显示出来。

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

下面是将对象（窗口）和信号处理函数连接起来的两个例子。在此处捕获了 `delete_event` 和 `destory` 两个信号。当我们用窗口管理程序关闭窗口时，或当我们在窗口的某个构件中调用 `gtk_widget_destroy()` 销毁窗口时，将引发第一个信号。在 `delete_event` 处理函数中返回 `FALSE` 时，将引发第二个信号。`GTK_OBJECT` 和 `GTK_SIGNAL_FUNC` 是用于执行类型转换和检查的宏，它们还提高了代码的可读性。

```
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
GTK_SIGNAL_FUNC (delete_event), NULL);
gtk_signal_connect (GTK_OBJECT (window), "destory",
GTK_SIGNAL_FUNC (destory), NULL);
```

下面这个函数用于设置放容器对象的属性。这里将设置窗口的属性，让它内部没有构件

占据的位置有一个宽度为 10 像素的空白区域。在这一节我们还会看到其他一些类似的设置构件属性的函数。其中 GTK_CONTAINER 是用于类型转换的宏。

```
gtk_container_set_border_width (GTK_CONTAINER (window), 10);
```

下面的函数将创建一个新按钮。它在内存中为一个新的 GtkWidget 结构类型分配空间，然后对它进行初始化，并且让按钮指针指向它。按钮显示时，它上面显示 “Hello World” 标签。

```
button = gtk_button_new_with_label ("Hello World");
```

在这里，我们创建了一个按钮，并且让它做点儿有用的事。我们在按钮上添加一个信号处理函数，当它引发 clicked 信号时，调用 hello() 函数。在这里我们不想向函数传递参数，因而我们简单传递一个 NULL 给 hello() 回调函数。很明显，当我们用鼠标点击按钮时就会引发 clicked 信号。

```
gtk_signal_connect (GTK_OBJECT (button), "clicked",  
GTK_SIGNAL_FUNC (hello), NULL);
```

我们也用这个按钮退出应用程序。这说明了 destroy 信号可以来自窗口管理程序，也可以来自于应用程序。

当按钮被按下后，同上面所述一样，它首先调用 hello() 回调函数，然后按设置的次序调用其他函数。根据需要，可以有多个回调函数。它们会依据它们的连接次序依次执行。因为 gtk_widget_destroy() 函数仅仅接受 GtkWidget *widget 作为参数，在这里我们用 gtk_signal_connect_object() 函数代替 gtk_signal_connect()。

```
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",  
GTK_SIGNAL_FUNC (gtk_widget_destroy),  
GTK_OBJECT (window));
```

下面的函数调用是一个组装调用，后面将会专门解释。它是相当容易理解的。它简单地告诉 GTK，按钮应该放置在窗口中，并且可以在窗口中显示。要注意，一个 GTK 容器构件仅仅能容纳一个子构件。还有一些其他构件，它们作为容器可以用多种方法容纳多个构件。

```
gtk_container_add (GTK_CONTAINER (window), button);
```

现在，我们已经拥有了创建一个应用程序所需要的所有方法。当设置了全部信号处理函数，并且将按钮放在窗口中恰当的地方后，我们要求 GTK 在屏幕上 “显示” 所有的构件。最好让窗口构件最后显示，这样整个窗口 - 包括里面的所有构件 - 将一起弹出来，而不是窗口先弹出来，然后窗口内的按钮再显示出来。不过，在这个简单的例子里，很难注意到这种区别。

```
gtk_widget_show (button);  
gtk_widget_show (window);
```

然后，我们调用 gtk_main()，开始等候来自 X 服务器的事件发生。这些事件的发生，会使某个构件引发信号。

```
gtk_main ();
```

最后是 main() 函数的返回值。当调用 gtk_quit() 时，控制返回到这里。

```
return (0);
```

现在，当我们在 GTK 按钮上点击鼠标按键时，构件引发 clicked 信号。要使用这些信息，我们的程序中应设置一个信号处理函数以捕获这个信号。在本例子里，当我们创建的按钮被

“点击”时，将调用 `hello()` 函数，并给它传递一个 `NULL` 参数，然后调用这个信号的下一个处理函数。调用 `gtk_widget_destroy()` 函数，将窗口构件作为它的参数，运行结果是销毁窗口构件。这导致窗口引发 `destroy` 信号，当这个信号被捕获时，调用 `destroy()` 回调函数，简单退出 GTK。

另一种方法是用窗口管理程序关闭窗口时，会引发 `delete_event` 事件。它调用 `delete_event` 处理函数。如果返回 `TRUE`，将保留窗口，就像什么也没有发生一样。返回 `FALSE` 将导致 GTK 引发 `destroy` 信号，调用 `destroy` 信号的回调函数，退出 GTK。

2.2.6 运行helloworld

上面介绍了 `helloworld` 中代码的含义。编译完成后，在应用程序所在目录下输入以下命令，运行 `helloworld`（结果如图 2-2 所示）：

```
./helloworld
```

尝试一下，将鼠标放在窗口的边框处，按下鼠标左键，拖动鼠标以改变窗口大小，按钮会随之而改变大小。点击“Hello World”按钮会退出应用程序。



图2-2 Gtk 版本的“Hello World”

2.3 Gnome应用程序

上面我们通过两个例子介绍了使用 Gtk+ 库创建 Linux 应用程序的步骤。实际上，如果要用到 Gnome 库，在代码中只有一些细微的区别。

首先，应该在代码的头部包含 `gnome.h` 而不是 `gtk.h`：

```
#include <gnome.h>
```

`gnome.h` 头文件中已经包含了 `gtk.h` 文件。

其次，应该用 `gnome_init()` 函数替换 `gtk_init()` 函数。`Gnome_init()` 的声明如下：

```
gnome_init(const char* app_id,  
           const char* app_version,  
           int argc,  
           char** argv)
```

`gnome_init()` 函数会在内部调用 `gtk_init()`。`gnome_init()` 函数的第一个参数是应用程序的名称，第二个参数是代表应用程序版本的字符串。这些参数是 Gnome 库内部使用的（例如，由参数分析程序提供的一些缺省信息）。

与 `gtk_init()` 函数一样，`gnome_init()` 分析命令行参数；但是与 `gtk_init()` 不一样的是，它不会改变 `argc` 和 `argv` 的值。如果想分析特定的选项，应该使用 `gnome_init_with_popt_table()` 函数。我们会在后面的内容中介绍这个函数。

如果初始化失败，`gnome_init()` 函数的返回值应该是一个非 0 的值，但在当前的 Gnome 版本中它总返回 0（如果有什么问题，比如说没有找到 X 服务器，`gnome_init()` 会简单地异常终止）。通常的惯例是忽略它的返回值，至少在 Gnome 1.0 中是这样的，但是，无论如何，检查它的返回值是一个好主意，这样可以防止未来的 Gnome 版本返回一个错误。

最后，在 Gnome 应用程序中所有用户可见的字符串都应该为翻译做标记。翻译是用 GNU 的 `gettext` 实用程序完成的，这称为“国际化”。在程序的开头，必须调用 `bindtextdomain()` 和

textdomain() 函数。这两个函数的调用形式如下：

```
bindtextdomain (PACKAGE, PACKAGE_LOCALE_DIR);
textdomain (PACKAGE);
```

对国际化问题，我们将在后面另辟专题进行介绍。

除了上面的几点以外，使用 Gnome库和Gtk库没有什么两样，比如创建构件、连接信号和回调函数、主循环等等。

2.4 GNU C 编译器

目前最常用的GNU C编译器(gcc)是一个全功能的ANSI C兼容编译器。如果熟悉其他操作系统或硬件平台上的一种C编译器，将能很快地掌握 gcc。gcc可以用于编译C语言和用C++语言编的代码，不管它使用的是什么函数库和构件库。下面简要介绍如何使用 gcc 和一些 gcc 编译器最常用的选项。

2.4.1 使用 gcc

通常后跟一些选项和文件名来使用 gcc编译器。gcc 命令的基本用法如下：

```
gcc [options] [filenames]
```

命令行选项指定的操作将在命令行上每个给出的文件上执行。下面将介绍一些最常用到的选项。

2.4.2 gcc 选项

gcc 有超过100个的编译选项可用。这些选项中的许多选项一般根本不会用到，但一些主要的选项将会频繁用到。很多的 gcc选项包括一个以上的字符。因此必须为每个选项指定各自的连字符，并且就像大多数 Linux命令一样，不能在一个单独的连字符后跟一组选项。例如，下面的两个命令是不同的：

```
gcc -p -g test.c
gcc -pg test.c
```

第一条命令告诉 gcc编译test.c时为prof命令建立剖析信息并且把调试信息加入到可执行的文件里。第二条命令只告诉 gcc为gprof命令建立剖析信息。

如果编译一个程序时不使用任何选项， gcc 将会建立(假定编译成功)一个名为 a.out 的可执行文件。例如，下面的命令将在当前目录下产生一个名为 a.out的文件：

```
gcc test.c
```

可以使用-o编译选项来为即将产生的可执行文件指定一个文件名来代替 a.out。例如，将一个叫count.c的C程序编译为名叫count的可执行文件，可以输入下面的命令：

```
gcc -o count count.c
```

gcc 同样有指定编译器处理多少的编译选项。-c选项告诉 gcc 仅把源代码编译为目标代码而跳过汇编和连接的步骤。这个选项使用得非常频繁，因为它使编译多个 C程序时的速度更快并且更易于管理。缺省时 gcc建立的目标代码文件有一个.o的扩展名。

-S 编译选项告诉 gcc在为C代码产生了汇编语言文件后停止编译。 gcc 产生的汇编语言文件的缺省扩展名是.s。-E选项指示编译器仅对输入文件进行预处理。当使用这个选项时，预处

理器的输出被送到标准输出而不是储存在文件里。

1. 优化选项

用gcc编译C代码时，它会尝试用最少的时间完成编译并且使编译后的代码易于调试。易于调试意味着编译后的代码与源代码有同样的执行次序，编译后的代码没有经过优化。有很多选项可用于告诉 gcc 在耗费更多编译时间和牺牲易调试性的基础上产生更小更快的可执行文件。这些选项中最典型的是 -O和-O2选项。

-O选项告诉gcc对源代码进行基本优化。这些优化在大多数情况下都会使程序执行得更快。-O2选项告诉gcc产生尽可能小和尽可能快的代码。-O2选项将使编译的速度比使用-O时慢。但通常产生的代码执行速度会更快。

除了-O和-O2优化选项外，还有一些低级选项用于产生更快的代码。这些选项非常特殊，而且最好只有完全理解这些选项将会对编译后的代码产生什么样的效果时再去使用它们。

2. 调试和剖析选项

gcc 支持数种调试和剖析选项。在这些选项里最常用到的是 -g和-pg选项。

-g选项告诉gcc产生能被GNU调试器gdb使用的调试信息以便调试程序。gcc提供了一个很多其他C编译器里没有的特性，在gcc里可以将-g和-O (产生优化代码)联用。这一点非常有用，因为这样可以在与最终产品尽可能相近的情况下调试代码。在同时使用这两个选项时必须清楚所写的某些代码已经在优化时被gcc做了改动。

-pg选项告诉gcc在程序里加入额外的代码，执行时，将产生 gprof用的剖析信息以显示程序的耗时情况。

上面介绍的都是关于 gcc的最简单的知识。如果想详细了解 gcc，请参考GCC-HOWTO，或者在shell提示符下输入：

```
man gcc
```

这样可以浏览gcc的手册页。

当然，对规模较大的程序来说，仅仅一行编译指令是远远不够的。有多种 GNU实用工具可以帮助完成编译选项的设置，如 autoconf、automake和libtool等。GUI生成器Glade所生成的代码中有一个autogen.sh脚本，也可以用来完成这件工作。

2.5 初始化库

前面已经介绍过，在程序的 main函数一开始，应用程序必须调用 gtk_init()函数初始化 Gtk+库。对Gnome应用程序，要用gnome_init()函数代替gtk_init()函数(gnome_init()会在内部调用gtk_init())。

gnome_init()函数的第一个参数是应用程序的名称，第二个参数是代表应用程序版本的字符串。这些参数是Gnome库内部使用的(例如，由参数分析程序提供的一些缺省信息)。

函数列表：初始化Gnome

```
#include <libgnomeui/gnome-init.h>
int gnome_init(const char* app_id,
               const char* app_version,
               int argc,
               char** argv)
```

与gtk_init()函数一样，gnome_init()分析命令行参数；但是与gtk_init()不一样的是，它不

会改变argc和argv的值。如果想分析特定的选项，应该使用 `gnome_init_with_popt_table()` 函数。

如果初始化失败，`gnome_init()` 函数的返回值应该是一个非 0 的值，但目前它总返回 0 (如果有什么问题，比如说没有找到 X 服务器，`gnome_init()` 会简单地异常终止)。通常的习惯是忽略它的返回值。但是，无论如何，检查它的返回值是一个好主意，这样可以防止未来的 Gnome 版本产生错误。

2.6 用popt分析参数

对基于 Gnome 的应用程序，如果允许用户从命令行带参数启动，应该在初始化的时候对传递到应用程序的参数和选项进行分析。

2.6.1 参数分析方法

Gnome 使用一个强劲的选项分析库 `popt` 来实现这一点。`popt` 处理所有的缺省 Gnome 选项。要查看 Gnome 的缺省选项，只需将 `--help` 选项传递给任何 Gnome 应用程序即可。还可以用定制的选项添加一个 “`popt` 表”。要做到这一点，用 `gnome_init_with_popt_table()` 函数代替 `gnome_init()` 函数。

函数列表：初始化库，进行参数分析

```
#include <libgnomeui/gnome-init.h>
int gnome_init_with_popt_table(const char* app_id,
                               const char* app_version,
                               int argc,
                               char** argv,
                               const struct poptOption* options,
                               int flags,
                               poptContext* return_ctx)
```

`popt` 表是一个 `poptOption` 结构数组。下面是其定义：

```
struct poptOption {
    const char* longName;
    char shortName;
    int argInfo;
    void* arg;
    int val;
    char* descrip;
    char* argDescrip;
};
```

下面解释 `poptOption` 结构中各成员的含义。

前面的两个部分 `longName` 和 `shortName` 是选项的长名字和短名字，例如 “`help`” 和 “`h`” 对应于命令行选项的 `help` 和 `-h`。如果只想有一个选项名，它们可以设为 `NULL` 和 `'\0'`。

第三个成员 `arginfo` 告知表输入项是什么类型。可以是以下几种取值：

- `POPT_ARG_NONE` 表明选项只是一个简单的开关，它没有参数。
- `POPT_ARG_STRING` 表明选项有一个字符串参数，比如说 `--geometry="300 x 300+50+100"`。
- `POPT_ARG_INT` 表明选项带一个整数的参数，例如 `--columns=10`。

- POPT_ARG_LONG 表明选项带一个长整型的参数。
- POPT_ARG_INCLUDE_TABLE表明poptOption结构不指定一个选项，但是要包含另一个popt表。
- POPT_ARG_CALLBACK表明poptOption结构并不指定选项，而是一个分析表中选项的回调函数。入口种类应该在表的开头。
- POPT_ARG_INTL_DOMAIN表明poptOption结构指定这个表和任何子表的翻译范围。

arg的意义依赖于arginfo成员。对一个带参数的选项来说，arg应该指向一个参数类型的变量。popt会用参数填充所指向的变量。对 POPT_ARG_NONE，如果选项能在命令行上找到，*arg设置为TRUE。对所有情况，arg都可以设置为NULL，popt会忽略它。

对POPT_ARG_INCLUDE_TABLE，arg指向所包含的表；对 POPT_ARG_CALLBACK，它指向要调用的回调函数；对 POPT_ARG_INTL_DOMAIN，它应该是翻译范围字符串。

val成员是每个成员的标识符。它在 Gnome应用程序中一般没有什么用，但是如果使用一个回调函数，则它在回调函数中是有用的。如果不想用它，把它设置为 0。

最后两个成员用于对 --help选项自动生成输出信息。descrip用于描述选项；argDescrip 用于描述选项的参数。例如，--display选项的帮助是这个样子：

```
--display=DISPLAY          X display to use
```

在这里，argDescrip是“DISPLAY”；descrip是“X display to use”。要用_宏为这两个字符串标记以便翻译。

对POPT_ARG_INCLUDE_TABLE，descrip意义略有不同。在这种情况下，它在帮助输出中为一“组”选项加标题。例如，在下面输出中的“帮助选项”：

帮助选项

-?, --help 显示帮助信息

--usage 显示简要的用法消息

如果在popt表的开头放一个POPT_ARG_CALLBACK 类型条目，会对命令行中的每个选项的信息调用一个用户定义的回调函数。下面是回调函数应该有的类型：

```
typedef void (*poptCallbackType)(poptContext con,
                                  enum poptCallbackReason reason,
                                  const struct poptOption* opt,
                                  const char* arg,
                                  void* data);
```

poptContext对象是不透明的，它包含所有的 popt状态。这样就有可能在同一个程序中多次使用popt，或者同时分析一套以上的选项。还可以用由 popt提供的函数从poptContext中提取出当前分析状态的信息。

poptCallbackReason可取以下值：

- POPT_CALLBACK_REASON_PRE
- POPT_CALLBACK_REASON_POST
- POPT_CALLBACK_REASON_OPTION

回调函数将POPT_CALLBACK_REASON_OPTION 作为“reson”参数，对命令行中的每个选项调用一次。

根据要求，也可以在参数分析之前或之后调用。在这些情况下，“reson”参数会是

POPT_CALLBACK_REASON_PRE或 POPT_CALLBACK_REASON_POST。要指定回调函数在参数分析之前或之后调用，必须将上面两个标志与 POPT_ARG_CALLBACK 结合起来使用。例如，下面的poptOption结构初始化程序指定在参数分析之前和之后都调用回调函数：

```
{ NULL, '\0', POPT_ARG_CALLBACK|POPT_CBFLAG_PRE|POPT_CBFLAG_POST,
  &parse_an_arg_callback, 0, NULL}
```

回调函数的opt参数就是对应于最近看见的命令行选项的 poptOption结构。可以访问这个结构的val成员以判定只进行查找的选项是哪一个。arg参数是传递到命令行选项的任何参数的原文；data参数是回调函数的数据，这个数据就是在回调函数里指定的 popyOption结构的descrip成员。

在Gnome环境中gnome_init_with_popt_table()的flags参数基本上可以忽略掉，这个“flags”用处不大。

如果给gnome_init_with_popt_table()函数的最后一个参数return_ctx传递一个非空的指针，会返回当前的环境。可以用这个环境来提取那些不是选项的部分，比如文件名。这是使用poptGetArgs()完成的。下面是一个例子：

```
char** args;
poptContext ctx;
int i;

bindtextdomain (PACKAGE, GNOMELOCALEDIR);
textdomain (PACKAGE);
gnome_init_with_popt_table(APPNAME, VERSION, argc, argv,
                          options, 0, &ctx);

args = poptGetArgs(ctx);

if (args != NULL)
{
    i = 0;
    while (args[i] != NULL)
    {
        /* Do something with each argument */
        ++i;
    }
}

poptFreeContext(ctx);
```

注意，必须释放前面使用过的 poptContext变量。然而，如果对 return_ctx传递一个NULL值，库函数会释放它。还要记住，如果命令行没有参数，poptGetArgs()会返回NULL。

2.6.2 GnomeHello程序的参数分析

下面的示例代码来自于一个名为 GnomeHello的程序。这个程序中集中演示了 Gnome编程的各种技巧。GnomeHello的源代码见附录。

如果带有help启动程序，GnomeHello将输出下面的信息：

```
$ ./hello --help
```

```

Usage: hello [OPTION...]

GNOME Options
  --disable-sound           Disable sound server usage
  --enable-sound           Enable sound server usage
  --espeaker=HOSTNAME:PORT Host:port on which the sound server to use is
                           running

Help options
  -?, --help               Show this help message
  --usage                  Display brief usage message

GTK options
  --gdk-debug=FLAGS        Gdk debugging flags to set
  --gdk-no-debug=FLAGS     Gdk debugging flags to unset
  --display=DISPLAY        X display to use
  --sync                   Make X calls synchronous
  --no-xshm                Disable X shared memory extension
  --name=NAME              Program name as used by the window manager
  --class=CLASS            Program class as used by the window manager
  --gxid_host=HOST
  --gxid_port=PORT
  --xim-preedit=STYLE
  --xim-status=STYLE
  --gtk-debug=FLAGS        Gtk+ debugging flags to set
  --gtk-no-debug=FLAGS     Gtk+ debugging flags to unset
  --g-fatal-warnings       Make all warnings fatal
  --gtk-module=MODULE      Load an additional Gtk module

GNOME GUI options
  -V, --version

Help options
  -?, --help               Show this help message
  --usage                  Display brief usage message

Session management options
  --sm-client-id=ID        Specify session management ID
  --sm-config-prefix=PREFIX Specify prefix of saved configuration
  --sm-disable             Disable connection to session manager

GnomeHello options
  -g, --greet              Say hello to specific people listed on the
                           command line
  -m, --message=MESSAGE    Specify a message other than "Hello, World!"
  --geometry=GEOMETRY       Specify the geometry of the main window
$

```

对所有 Gnome 应用程序来说，所有这些选项差不多都是相同的，只有最后三个，标为“GnomeHello options”的是 GnomeHello 特有的。--greet 或 -g 选项打开“问候模式”；GnomeHello 期望在命令行上输入姓名列表，创建一个对话框向输入的每一个名字打招呼。--message 选项期望输入一个字符串参数代替常见的“Hello, World!”消息；--geometry

选项期待一个标准的 X 几何字符串，指定主窗口的尺寸和位置。

下面是 GnomeHello 用作参数分析的变量和 popt 表：

```
static int greet_mode = FALSE;
static char* message = NULL;
static char* geometry = NULL;
struct poptOption options[] = {
    {
        "greet",
        'g',
        POPT_ARG_NONE,
        &greet_mode,
        0,
        N_("Say hello to specific people listed on the command line"),
        NULL
    },
    {
        "message",
        'm',
        POPT_ARG_STRING,
        &message,
        0,
        N_("Specify a message other than \"Hello, World!\""),
        N_("MESSAGE")
    },
    {
        "geometry",
        '\0',
        POPT_ARG_STRING,
        &geometry,
        0,
        N_("Specify the geometry of the main window"),
        N_("GEOMETRY")
    },
    {
        NULL,
        '\0',
        0,
        NULL,
        0,
        NULL,
        NULL
    }
};
```

下面是 main() 的第一部分，在这里 GnomeHello 检验参数是否已经正确组合并装配成一个要欢迎的人员名单列表：

```
GtkWidget* app;
poptContext pctx;
char** args;
int i;
```

```
GSList* greet = NULL;
GnomeClient* client;

bindtextdomain(PACKAGE, GNOMELOCALEDIR);
textdomain(PACKAGE);

gnome_init_with_popt_table(PACKAGE, VERSION, argc, argv,
                           options, 0, &pctx);

/* Argument parsing */

args = poptGetArgs(pctx);

if (greet_mode && args)
{
    i = 0;
    while (args[i] != NULL)
    {
        greet = g_slist_prepend(greet, args[i]);
        ++i;
    }
    /* Put them in order */
    greet = g_slist_reverse(greet);
}
else if (greet_mode && args == NULL)
{
    g_error(_("You must specify someone to greet."));
}
else if (args != NULL)
{
    g_error(_("Command line arguments are only allowed with --greet."));
}
else
{
    g_assert(!greet_mode && args == NULL);
}

poptFreeContext(pctx);
```

2.7 国际化

在Gnome应用程序中所有用户可见的字符串都应该为翻译做标记。翻译是用 GNU的gettext实用程序完成的。gettext只是一个简单的消息分类，它存储了键/值对，键是程序的硬编码字符串，值是翻译过的字符串（如果有的话），或者只有键（如果没有翻译，或者键的语言已经是正确的）。

作为程序员，不一定要提供软件的多国语言版本。不过，强烈建议将其中的字符串作为翻译标记，这样，gettext脚本能够提取出一个要翻译的字符串表。程序的用户可以根据他的实际情况决定是否编译一个对应于当前语言的版本。

宏列表：翻译宏

```
#include <libgnome/gnome-i18n.h>
_(string)
N_(string)
```

Gnome使用上面的两个宏来实现翻译标记。宏 `_()` 对字符串做翻译标记的同时还进行信息类别查找。应该在任何 C 准许函数调用的环境中使用它。宏 `N_()` 不做后一项操作，仅仅是为字符串做翻译标记。可以在不允许进行函数调用的场合使用它，例如在静态数组初始化程序中。如果使用宏 `N_()` 为字符串做翻译标记，最后必须对它调用宏 `_()` 以做实际的查找。

下面是一个简单的例子：

```
#include <gnome.h>
static char* a[] =
    N_("Translate Me"),
    N_("Me Too")
};

int main(int argc, char** argv)
{
    bindtextdomain(PACKAGE, GNOMELOCALEDIR);
    textdomain(PACKAGE);
    printf(_("Translated String\n"));
    printf(_(a[0]));
    printf(_(a[1]));
    return 0;
}
```

注意到字符串“Translate Me”和“Me Too”已经做了标志，这样，`gettext`就能发现它们，并产生一个要翻译的字符串列表。翻译程序将用这个表生成实际的翻译字符串。以后，宏 `_()` 包含了一个函数调用，用以对数组的每个成员进行翻译查找。因为当字符串文字“Translated String”引入时允许进行函数调用，所有的工作都是在一步内完成的。

在程序的开头，必须调用 `bindtextdomain()` 和 `textdomain()` 函数，如同上面的例子显示的那样。在上面的代码中，`PACKAGE` 是一个字符串，它代表程序找到的字符串包，典型情况下它是在 `config.h` 中定义的。还必须定义 `GNOMELOCALEDIR` 目录，典型情况下它是在 `Makefile.am` 中定义的（标准值应该是 `$(prefix)/share/locale`，或 `$(datadir)/locale`）。翻译是存储在 `GNOMELOCALEDIR` 路径下的。

当用字符串为翻译做标记时，必须保证字符串是可翻译的。要避免在运行时通过连接多个字符串生成一个字符串。例如，不要像下面这样做：

```
gchar* message = g_strconcat(_("There is an error on device "),
                               device, NULL);
```

问题是：在一些语言中，应该将设备名称放在前面（或放在中间）。如果使用 `g_snprintf()` 或者 `g_strdup_printf()` 函数而不是字符串连接函数，翻译程序就能够改变单词的次序。下面就是正确的方法：

```
gchar* message = g_strdup_printf(_("There is an error on device %s"),
                                   device);
```

现在，翻译程序能够根据需要移动 `%s` 的位置。

应该尽可能避免复杂的语法。例如，要翻译下面的字符串就很困难：

```
printf(_("There %s %d dog%s\n"),
        n_dogs > 1 ? _("were") : _("was"),
        n_dogs,
        n_dogs > 1 ? _("s") : "");
```

最好将条件移动到printf()的外面：

```
if (n_dogs > 0)
    printf(_("There were %d dogs\n"), n_dogs);
else
    printf(_("There was 1 dog\n"));
```

然而，即使这样也不一定能够正常工作。一些语言还有比“正好一个”和“一个以上”更多的分类(也就是，在英语的单复数形式以外，它们还有“正好两个”的说法)。可以使用以下做法：

```
static const char* ndogs_phrases[] = {
    N_("There were no dogs.\n"),
    N_("There was one dog.\n"),
    N_("There were two dogs.\n"),
    N_("There were three dogs.\n")
};
```

可是这样处理也太麻烦了。如果可能应该尽量避免出现这种情况。

当分析和显示特定种类的数据（包括日期和十进制数）时，也必须考虑国际化。通常，C库函数提供了足够的实用函数以处理这些情况，使用strftime()、strcoll()等函数处理这种情况。GlibGDate的实用函数在内部也是用strftime()函数处理日期数据的。

应避免的常见错误：当读和写文件时，不要使用依赖于特定地区的函数。例如，printf()和scanf()函数根据地区调整它们的小数位格式，所以不能在文件中使用这种格式。如果这样，欧洲的用户将不能读出在美国常见的文件。

2.8 保存配置信息

有时候，需要保留一些应用程序的配置信息。例如，在文件菜单里保存一个“最近打开的文件”列表，应用程序是否显示工具条、状态条等。这些功能都可以使用Gnome API函数实现。不过，有些设置值，比如上次窗口打开的位置、尺寸等，一般不这么处理，而是使用会话管理功能实现。

libgnome库有在普通文本配置文件里保存简单的键/值对的能力。提供的实用程序能够处理数值型和布尔型的值，可以透明地将变量值转换为文本文件，或者相反，从文本文件读出。Gnome配置文件的标准位置是~/.gnome，库函数文件用这个位置作为缺省位置。不过，库函数也可以使用任何其他文件。每个函数还有相应的变体函数，它们能将配置文件存到~/.gnome_private下，这个目录具有用户权限设置。通常也将这个libgnome模块称为gnome-config。不要将这个gnome-config与gnome-config脚本混淆，后者是Gnome程序用来报告编译和链接标志的。

gnome-config函数用路径作为参数。路径由三部分组成：

- 要用到的文件名，在~/.gnome或~/.gnome_private目录下。按惯例是应用程序的名字。
- 一个节，相关配置信息的逻辑子类。
- 一个键，键/值对的一半。键实际上是与一块配置数据相联系的。

路径作为一个字符串传给Gnome，并使用“/filename/section/key”的形式。如果想用一个不在标准Gnome目录下的文件名，可以将整个路径用“=”隔开，它将被解释为绝对路径。甚至可以将配置文件作为简单的数据文件格式（可以用作.desktop文件——设置在里面的文件会出现在Gnome的主菜单上）。然而，XML（也许要使用gnome-xml软件包）才是这种情况的更好的选择。对存储某些种类的配置信息，XML也许是一个更好的选择，libgnome配置函数库的主要优点就是简单。

gnome-config已经有很长的历史了，它最初是为WINE——Windows仿真器项目写的，然后用在GNU Midnight Commander文件管理器上，最后移植到Gnome库上。当前的计划是在下一个Gnome版本中用更强劲的库函数取代gnome-config，主要想支持按主机配置、LDAP（轻量级目录存取协议）后端，以及一些其他特性。然而，即使下层的引擎（函数库）发生剧烈变化，gnome-config API函数也总是会得到支持的。

2.8.1 读出存储的配置数据

从文件中获得数据很简单，只要调用一个函数获取给定键的值就可以了。取值的函数接受一个路径作为参数。例如，你可能询问用户是否想要看到一个对话框：

```
gboolean show_dialog;  
show_dialog =  
    gnome_config_get_bool("/myapp/General/dialog");
```

如果配置文件还不存在，或没有键名与你提供的键匹配，函数返回 0、FALSE或NULL。返回字符串的函数会返回一段分配的内存，应该用 g_free()函数将字符串释放。字符串矢量函数返回一个已分配空间的字符串数组（释放该矢量的最容易的方法是调用 g_strfreev()函数）。

如果给定键不存在，可以指定一个缺省值，在路径后加一个“=value”。例如：

```
gboolean show_dialog;  
show_dialog =  
    gnome_config_get_bool("/myapp/General/dialog=true");
```

每个函数都有一个 with_default式样的变体，这些函数告诉你返回的值是从配置文件取得的还是从指定的缺省值取得的。例如：

```
gboolean show_dialog;  
gboolean used_default;  
show_dialog =  
    gnome_config_get_bool_with_default("/myapp/General/dialog=true",  
                                       &used_default);  
  
if (used_default)  
    printf("Default value used for show_dialog\n");
```

gnome_config_push_prefix()和gnome_config_pop_prefix()函数可以用于避免每次都要指定整个路径。例如：

```
gboolean show_dialog;  
gnome_config_push_prefix("/myapp/General/");  
show_dialog = gnome_config_get_bool("dialog=true");  
gnome_config_pop_prefix();
```

这些函数在保存值时也起作用。

在名字中带一个private的配置函数使用具有权限限制的.gnome_private目录，如上面所讨论的。带translated_string后缀的函数限定用当前场所的名字限定给定的键，这些函数一般用

函数列表：从配置文件获取数据

[illegible]

```
gchar*** argvp,  
gboolean* was_default)
```

2.8.2 在配置文件中存储数据

保存数据是与获取数据相反的过程，用同样的方法给出一个路径“ /file/section/key ”，连带要存储的值。数据并不是立即写入的，必须调用 `gnome_config_sync()` 函数以保证文件写到磁盘上了。

函数列表：保存数据到配置文件

```
#include <libgnome/gnome-config.h>  
void gnome_config_set_string(const gchar* path,  
                             const gchar* value)  
void gnome_config_set_translated_string(const gchar* path,  
                                         const gchar* value)  
  
void gnome_config_set_int(const gchar* path,  
                          gint value)  
  
void gnome_config_set_float(const gchar* path,  
                            gdouble value)  
  
void gnome_config_set_bool(const gchar* path,  
                           gboolean value)  
  
void gnome_config_set_vector(const gchar* path,  
                             gint argc,  
                             const gchar* const argv[])  
  
void gnome_config_private_set_string(const gchar* path,  
                                     const gchar* value)  
  
void gnome_config_private_set_translated_string(const gchar* path,  
                                                 const gchar* value)  
  
void gnome_config_private_set_int(const gchar* path,  
                                  gint value)  
  
void gnome_config_private_set_float(const gchar* path,  
                                    gdouble value)  
  
void gnome_config_private_set_bool(const gchar* path,  
                                   gboolean value)  
  
void gnome_config_private_set_vector(const gchar* path,  
                                     gint argc,  
                                     const gchar* const argv[])
```

2.8.3 配置文件迭代器

迭代器用于在给定文件中扫描节，或在给定节中扫描键。应用程序可以用这个特性存储

数据列表。具体实现方法是通过动态生成键或节的名字以保存数据，随后迭代它们以检查保存了些什么。

迭代器是一种不透明的数据类型；可以将“节”的名称传给 `gnome_config_init_iterator()` 函数，迭代一遍，并依次接收一个迭代器。然后调用 `gnome_config_iterator_next()` 函数从该节中获得键/值对。从 `gnome_config_iterator_next()` 函数返回的键/值对必须用 `g_free()` 函数释放，`gnome_config_iterator_next()` 函数返回的值是一个指向下一个迭代器的指针。当函数 `gnome_config_iterator_next()` 返回 NULL 时，说明已经遍历了所有的键/值对。

gnome-apt程序中的迭代示例

下面是一个来自 `gnome-apt` 的迭代示例，在 Debian 发布版本中用来管理软件包的 C++ 应用程序。`gnome-apt` 在一个树状视图中保存和加载一些栏的位置。栏是用 `GAptPkgTree::ColumnType` 枚举类型标识的。`GAptPkgTree::ColumnTypeEnd` 是栏枚举类型的最后一个元素，它等于有效栏类型的数目。

```
static void
load_column_order(vector<GAptPkgTree::ColumnType> & columns)
{
    gpointer config_iterator;
    guint loaded = 0;

    config_iterator = gnome_config_init_iterator("/gnome-apt/ColumnOrder");

    if (config_iterator != 0)
    {
        gchar * col, * pos;
        columns.reserve(GAptPkgTree::ColumnTypeEnd);

        loaded = 0;
        while ((config_iterator =
                gnome_config_iterator_next(config_iterator,
                                           &col, &pos)))
        {
            // shouldn't happen, but 'm paranoid
            if (pos == 0 || col == 0)
            {
                if (pos) g_free(pos);
                if (col) g_free(col);
                continue;
            }

            GAptPkgTree::ColumnType ct = string_to_column(col);

            gint index = atoi(pos);

            g_free(pos); pos = 0;
            g_free(col); col = 0;

            // the user could mangle the config file to make this happen
            if (static_cast<guint>(index) >= columns.size())
```



```

        continue;

        columns[index] = ct;

        ++loaded;
    }
}

if (loaded != static_cast<guint>(GAptPkgTree::ColumnTypeEnd))
{
    // Either there was no saved order, or something is busted - use
    // default order
    columns.clear();

    int i = 0;
    while (i < GAptPkgTree::ColumnTypeEnd)
    {
        columns.push_back(static_cast<GAptPkgTree::ColumnType>(i));
        ++i;
    }

    // Clean the section - otherwise an old entry could
    // remain forever and keep screwing us up in the future.
    gnome_config_clean_section("/gnome-apt/ColumnOrder");
    gnome_config_sync();
}

g_return_if_fail(columns.size() ==
                  static_cast<guint>(GAptPkgTree::ColumnTypeEnd));
}

```

下面是用于保存“列”位置的函数：

```

static void
save_column_order(const vector<GAptPkgTree::ColumnType> & columns)
{
    g_return_if_fail(columns.size() ==
                      static_cast<guint>(GAptPkgTree::ColumnTypeEnd));

    int position = 0;
    vector<GAptPkgTree::ColumnType>::const_iterator i = columns.begin();
    while (i != columns.end())
    {
        gchar key[256];
        g_snprintf(key, 255, "/gnome-apt/ColumnOrder/%s", column_to_string(*i));
        gchar val[30];
        g_snprintf(val, 29, "%d", position);
        gnome_config_set_string(key, val);

        ++position;
        ++i;
    }
}

```

```
}  
  
    gnome_config_sync();  
}
```

在这段代码中，将枚举值保存为字符串而不是整数值。`column_to_string()`和`string_to_column()`函数使用了一个简单的、由枚举值索引的栏名数组，用于来回转换。这么做有两个理由：在程序的未来版本中枚举值发生变化时，代码不会因此而发生故障，同时，它还使配置文件可以人工修改。

栏位置是用`gnome_config_set_string()`而不是用`gnome_config_set_int()`存储的。这是因为`gnome_config_iterator_next()`返回一个代表存储信息的字符串。更可能的情况是：`gnome_config_set_int()`函数将整数存储为`atoi()`函数能理解的字符串（它确实能理解），但是从技术上来说，API函数并未保证这一点。如果代码中使用了`gnome_config_set_int()`函数，它将从`gnome_config_iterator_next()`函数获得一个唯一的键，然后调用`gnome_config_get_int()`函数获得整数值。对获得的字符串值使用`atoi()`函数会对`gnome-config`的实现产生没有根据的假设。

2.8.4 节迭代器

`gnome_config_init_iterator_sections()`允许在一个文件中迭代所有的节，而不是在一节中迭代键。迭代节时，`gnome_config_iterator_next()`函数忽略它的值参数并将节名放在键参数位置上。

函数列表：配置文件迭代器

```
#include <libgnome/gnome-config.h>  
void* gnome_config_init_iterator(const gchar* path)  
void* gnome_config_private_init_iterator(const gchar* path)  
void* gnome_config_init_iterator_sections(const gchar* path)  
void* gnome_config_private_init_iterator_sections(const gchar* path)  
void* gnome_config_iterator_next(void* iterator_handle,  
                                gchar** key,  
                                gchar** value)
```

2.8.5 其他的配置文件操作

下面的函数列表列举了一些可用于操作配置文件的其它操作函数。这些函数中最重要的已经介绍过了。`gnome_config_sync()`将配置文件写到磁盘上，`gnome_config_push_prefix()`允许缩短传递到其他`gnome-config`函数中的路径长度。还有一些布尔测试函数，用于询问`gnome-config`给定的节是否存在。

这里面有两个新函数。“删除”一个文件或节意味着忘掉任何存储在内存中的相关信息，包括从文件加载的缓存值和还没有用`gnome_config_sync()`函数保存到磁盘上的值。要“清除”一个文件、节或键意味着将它的值清除，所以，一旦调用`gnome_config_sync()`函数，文件、节或键都不再存在。

`gnome_config_sync()`函数自动调用`gnome_config_drop_all()`函数并释放所有的`gnome-config`资源，因为信息已经安全地在磁盘上存在了。

还有可以从gnome-config路径取得一个配置文件的实际(文件系统级)路径的函数。这些在应用程序中没有多大用处。

函数列表：其他配置文件函数

```
#include <libgnome/gnome-config.h>
gboolean gnome_config_has_section(const gchar* path)
gboolean gnome_config_private_has_section(const gchar* path)
void gnome_config_drop_all()
void gnome_config_sync()
void gnome_config_sync_file(const gchar* path)
void gnome_config_private_sync_file(const gchar* path)
void gnome_config_drop_file(const gchar* path)
void gnome_config_private_drop_file(const gchar* path)
void gnome_config_clean_file(const gchar* path)
void gnome_config_private_clean_file(const gchar* path)
void gnome_config_clean_section(const gchar* path)
void gnome_config_private_clean_section(const gchar* path)
void gnome_config_clean_key(const gchar* path)
void gnome_config_private_clean_key(const gchar* path)
gchar* gnome_config_get_real_path(const gchar* path)
gchar* gnome_config_private_get_real_path(const gchar* path)
void gnome_config_push_prefix(const gchar* path)
void gnome_config_pop_prefix()
```

2.9 会话管理

术语“会话”是指用户桌面状态的快照：哪个应用程序是打开的，窗口放在桌面的什么地方，每个应用程序打开了什么窗口，这些窗口的尺寸是多大，打开了什么文档，当前鼠标光标的位置等等。用户能在退出前保存这些会话，并且在下次登录时尽可能地自动恢复上次的会话。要做到这一点，应用程序必须有能力和恢复其状态特征，这些状态特征不是由窗口管理器控制的。

当应用程序应该保存状态时，由一个称为会话管理器的特殊程序通知应用程序。Gnome桌面环境带有一个称为gnome-session的会话管理器，但是Gnome用的X会话管理规范已经过时好几年了。CDE(通用桌面环境)使用同样的规范，KDE(K桌面环境)也准备采用这个规范。一个通过Gnome接口实现了会话管理的应用程序应该在任何会话管理的桌面环境上正常工作。Gnome确实对基本的规范(特别是启动“优先级”)做了一些扩展，但是这些应该不会中断其他的会话管理器，并且KDE也可能会实现这些规范。

使用GnomeClient对象

Gnome将应用程序与原始的、随X系统自带的会话管理接口屏蔽开来。这是通过一个称为GnomeClient的GtkObject对象做到的。GnomeClient代表应用程序与会话管理器的连接。Gnome管理会话管理的大多数细节。对大多数应用程序来说，只需要对两个请求响应。

- 当保存一个会话时，会话管理器会要求每个客户保存足够的信息，以便在下次登录时恢复状态。应用程序应该保存尽可能多的状态：当前打开的文档、鼠标光标的位置、命令历史等等。应用程序不用保存当前窗口的几何参数，窗口管理器会做这些事。

- 有时，会话管理器会要求客户关闭并退出（典型情况是当用户退出时）。当接收到这样的信息时，应该完成一些必要的工作，然后退出应用程序。

当会话管理器要求应用程序完成某个动作时，GnomeClient对象引发一个适当的信号。两个重要的信号是save_yourself和die。当应用程序保存它的状态时会引发 save_yourself信号，当应用程序应该退出时，引发 die信号。save_yourself信号的回调函数相当复杂，有好几个参数。die信号的回调函数很简单。

下面是来自于GnomeHello中的代码。GnomeHello获得一个指向GnomeClient对象的指针，并设置了一个信号回调函数：

```
client = gnome_master_client ();
gtk_signal_connect (GTK_OBJECT (client), "save_yourself",
                    GTK_SIGNAL_FUNC (save_session), argv[0]);
gtk_signal_connect (GTK_OBJECT (client), "die",
                    GTK_SIGNAL_FUNC (session_die), NULL);
```

argv[0]将用于save_yourself信号的回调函数。

下面是GnomeHello中的die回调函数：

```
static void
session_die(GnomeClient* client, gpointer client_data)
{
    gtk_main_quit ();
}
```

它的作用是退出应用程序。

下面是save_yourself的回调函数：

```
static gint
save_session (GnomeClient *client, gint phase, GnomeSaveStyle save_style,
              gint is_shutdown, GnomeInteractStyle interact_style,
              gint is_fast, gpointer client_data)
{
    gchar** argv;
    guint argc;
    /* allocate 0-filled, so it will be NULL-terminated */
    argv = g_malloc0(sizeof(gchar*)*4);
    argc = 1;
    argv[0] = client_data;

    if (message)
    {
        argv[1] = "--message";
        argv[2] = message;
        argc = 3;
    }
    gnome_client_set_clone_command (client, argc, argv);
    gnome_client_set_restart_command (client, argc, argv);

    return TRUE;
}
```

save_yourself信号必须告诉会话管理器怎样重新启动并克隆应用程序（创建一个新的实例）。重新启动的应用程序应该记住尽可能多的状态，在 GnomeHello例子中，它记住了显示的消息。保存应用程序状态最简单的方法是生成一个命令行，就像 GnomeHello所做的。可以要求GnomeClient使用gnome-config API函数做这些工作，还可以将信息保存在一个按会话配置的文件中。带有重要状态信息的应用程序需要使用这种方法。

2.10 Gtk+的主循环

Gtk+主循环的首要目的就是在连接到 X服务器的文件描述符上监听事件，并将事件转发到构件上。本节解释怎样使用主循环，怎样给主循环添加新功能：当主循环在指定的时间间隔内空闲时、当一个文件描述符已经读或写就绪、以及当主循环退出时调用一个函数。

2.10.1 主循环基本知识

从根本上来说，主循环是由 glib实现的。Gtk+将glib主循环连接到 Gdk的X服务器，并提供一个方便的接口（glib循环是比Gtk+的循环更低层的）。

gtk_main()函数运行主循环。直到调用 gtk_main_quit()函数，gtk_main()才会退出。gtk_main()函数可以递归调用，每次调用一个 gtk_main_quit()就退出gtk_main()函数的一个实例。gtk_main_level()函数返回递归的层次，也就是：如果没有 gtk_main()运行，返回0；如果一个gtk_main()函数在运行，返回1，等等。

gtk_main()函数的所有实例功能都是一样的，它们都监视同一个与 X服务器的连接，都对同样的事件队列起作用。gtk_main()实例用于阻塞、遮断一个函数的控制流直到满足某些条件。所有的Gtk+程序都用这个技巧使应用程序正在运行时main()函数不能退出去。gnome_dialog_run()函数使用了一个递归的主循环，此循环直到用户点击对话框的按钮时它才会返回。

有时候想处理一些事件，又不想将控制交给 gtk_main()，可以调用gtk_main_iteration()函数对主循环进行迭代。例如，这样可以处理单独的一个事件，它依赖于想将什么任务挂起。可以检查是否有任何事件需要通过调用 gtk_events_pending()函数处理。同样地，这两个函数允许临时将控制交还给 Gtk+。例如，在一个很长的计算中，想显示一个进度条，必须允许Gtk+主循环周期性地返回，让Gtk+能重画进度条。可以使用下面的代码：

```
while (gtk_events_pending())
    gtk_main_iteration();
```

下面是有关主循环的函数：

```
#include <gtk/gtkmain.h>
void gtk_main()
void gtk_main_quit()
void gtk_main_iteration()
gint gtk_events_pending()
guint gtk_main_level()
```

2.10.2 退出函数

退出函数就是当调用 gtk_main_quit()函数时要调用的回调函数。换句话说，回调函数只在

gtk_main()返回之前运行。回调函数应该是一个像下面这样定义的 GtkFunction：

```
typedef gint (*GtkFunction) (gpointer data);
```

退出函数是用 gtk_quit_add()添加进去的。添加退出函数时，必须指定一个由 gtk_main_level()返回的主循环的级别。第二个和第三个参数指定一个回调函数和回调数据。

回调函数的返回值说明了回调函数是否应该再次调用。只要回调函数返回 TRUE，它会被重复调用。只要它返回 FALSE，将取消与主循环的连接，并且不会再次调用。所有的退出函数都返回FALSE时，gtk_main()会返回。

gtk_quit_add()函数返回一个ID号码，可以用于用 gtk_quit_remove()函数删除该退出函数。还可以通过将回调数据传递给 gtk_quit_remove_by_data()函数来删除退出函数。

函数列表：退出函数

```
#include <gtk/gtkmain.h>
guint gtk_quit_add(guint main_level,
                  GtkFunction function,
                  gpointer data)
void gtk_quit_remove(guint quit_handler_id)
void gtk_quit_remove_by_data(gpointer data)
```

2.10.3 Timeout函数

有时候可能想应该在 gtk_main主循环中怎样让 GTK做点什么。这时可以创建一个定时 (Timeout) 函数，隔一定时间(毫秒)就调用一次。Timeout类似于其他编程环境中的定时器控件。下面的函数用于添加一个Timeout函数。

```
#include <gtk/gtkmain.h>
gint gtk_timeout_add( guint32      interval,
                    GtkFunction function,
                    gpointer      data );
```

第一个参数调用定时函数的时间间隔，以毫秒计。第二个参数是要调用的函数，第三个是要传递给函数的参数。函数返回一个整数值“标志”。可以用下面的函数停止调用定时函数：

```
#include <gtk/gtkmain.h>
void gtk_timeout_remove( gint tag );
```

其中tag参数是前一个函数返回的“标志”值。

还可以让回调函数返回FALSE或0来停止调用定时函数。也就是说，要想让函数继续调用，必须让它返回一个非0值或TRUE。

定期调用的回调函数声明应该是下面的形式：

```
gint timeout_callback( gpointer data );
```

可以看到，Timeout函数类似于许多可视化编程工具中的Timer控件（计时器）。

2.10.4 idle函数

当Gtk+主循环没有其他事情做时，idle函数连续运行。只有在事件队列是空的，并且主循环正常空闲着，正等待着有什么事情发生时，idle函数才会运行。只要它们返回 TRUE，这个函数就会一次又一次地调用；当它们返回 FALSE时，函数会被删除，就像调用了

gtk_idle_remove()函数一样。

列在下面函数列表中的 idle 函数 API，与 timeout 以及退出函数 API 是一样的。不过，不能在 idle 函数中调用 gtk_idle_remove() 函数，因为它会破坏 Gtk+ 的函数列表。要返回 FALSE 来删除 idle 函数。

idle 函数在对“只此一次”的代码排队时通常是很有用的，这样的代码一般在所有的事件已经处理之后才运行。相对昂贵（耗费系统资源较多）的操作，比如 Gtk+ 的大小协商以及 GnomeCanvas 重绘一般在返回 FALSE 的 idle 函数中发生。这保证了代价昂贵的操作只会执行一次，即使多个连续的事件独立地要求重新执行。

Gtk+ 的主循环包含一个简单的调度程序。idle 函数有一个分配给它们的优先级，就像 UNIX 进程所做的一样，也可以分配一个非缺省的优先级给 idle 函数。

函数列表：idle 函数

```
#include <gtk/gtkmain.h>
guint gtk_idle_add(GtkFunction function,
                  gpointer data)
void gtk_idle_remove(guint idle_handler_id)
void gtk_idle_remove_by_data(gpointer data)
```

2.10.5 输入函数

输入函数用于检查文件描述符的数据（由 open(2) 或 socket(2) 返回的文件描述符）。输入函数是在 Gdk 级处理的。当给定的文件描述符已经读写就绪时，会调用输入函数。它们对网络应用程序特别有用。关于文件描述符请参考 Unix 编程方面的参考书。

要添加一个输入函数，需要指定要监视的文件描述符、要等待的状态（读或写就绪），以及一个回调函数/数据对。下面的函数列表列出了这些 API。该函数可以用 gdk_input_add() 返回的标识符删除。不像 quit、timeout 和 idle 函数，从输入函数里面调用 gdk_input_remove() 函数删除该函数是安全的，Gtk+ 不会处于输入函数列表的迭代状态。

要指定等待的条件，使用 GdkInputCondition 标志：

GDK_INPUT_READ

GDK_INPUT_WRITE

GDK_INPUT_EXCEPTION

可以用 OR 将一个以上的标志连在一起。这些标志对应于传到 select() 系统调用的三种文件描述符集。要了解详细的内容，请参考 UNIX 的编程参考书。如果所有条件都得到满足，就会调用输入函数。

回调函数应该是这个样子：

```
typedef void (*GdkInputFunction) (gpointer data,
                                  gint source_fd,
                                  GdkInputCondition condition);
```

它接受回调数据、被监视的文件描述符以及要满足的条件（可能是一个正在监视的条件的子集）。

函数列表：输入函数

```
#include <gdk/gdk.h>
```



```
gint gdk_input_add(gint source_fd,  
                  GdkInputCondition condition,  
                  GdkInputFunction function,  
                  gpointer data)  
  
void gdk_input_remove(gint tag)
```

2.11 编译应用程序

用前面所介绍的基本概念，已经可以编译全功能的 Gtk+/Gnome应用程序了。但是还有一个大问题：如何配置编译选项？一些实用工具如 automake、autoconf、libtool等，可以用来简化这一过程。

为了方便维护，同时，也是为了便于使用这些实用工具，应该在编写代码时遵从一些约定。如果要程序发布为自由软件，最好能使程序源代码的目录结构遵从“GNU项目编码标准”。即使应用程序是私有的商用程序，不想公开源代码，从技术上来说，这么做也是一个非常好的选择，因为这些标准都是经过实践检验，能够让你节省大量的时间和精力。另外还应该在程序代码中包含INSTALL、README的文件。

2.11.1 生成源代码树

差不多所有的Gnome应用程序都使用同样的基于GNU工具automake、autoconf和libtool的编译系统。Gtk+和Gnome提供了一套autoconf宏，用于生成可移植的、符合标准的编译设置。我们用一个称为GnomeHello的应用程序来演示Gnome的特性。

Gnome应用程序遵从一系列的约定来生成源代码树和发布的tar文件，大多数约定被自由软件社区广泛使用。这些约定的许多方面已经在“GNU项目编码标准”(GNU Project's Coding Standards：http://www.gnu.org/prep/standards_toc.html)和Linux文件系统层次标准(Linux Filesystem Hierarchy Standard：<http://www.pathname.com/fhs/>)中正式化了。

GNU工具集，包括automake和autoconf使遵从这些标准变得很容易。然而，有时候你可能不想使用GNU工具集，例如，你也许需要一个统一的在Windows和MacOS平台上都能工作的编译工具(一些工具确实能在Windows平台上工作，它们使用Cygnus的“Cygwin”环境，参看<http://sourceware.cygnus.com/cygwin>)。

如果使用了autoconf和automake，除了编译应用程序，用户并不需要有这些工具。使用这些工具的目的是创建能在用户环境使用的、可移植的shell脚本和Makefile文件。

Autoconf实际上是一个工具集，其中包含aclocal、autoheader和autoconf等可执行文件。这些工具生成一个可移植的shell脚本——configure，configure和软件包一起发布给用户。它探查编译系统，生成Makefile文件和一个特殊的头文件config.h。由configure生成的文件能适应用户系统的特定环境。configure脚本从一个称为Makefile.in的模板文件生成每个Makefile文件。automake由一个手写的Makefile.am生成Makefile.in文件。Makefile.in文件随软件一同发布，当用户运行configure时会自动生成Makefile。

Libtool软件包是第三个重要的GNU工具，它的作用是确定共享库在特定平台上的特性。因为共享库在不同平台上可能会有所不同。

下面有一些Gnome软件包应该具有的特征：

- 一个README文件，介绍软件包。

- 一个INSTALL文件，解释怎样编译、安装软件包。
- 一个configure脚本，能使程序自动适应特定平台的特征（或者该平台所缺乏的特性）。configure可以带一个参数--prefix，指定要安装的软件包的位置。
- 标准的make目标，比如clean等等。
- 一个COPYING文件，包含软件包的版权信息。
- 一个ChangeLog文件，记录了软件的变化。
- 打包文件，一般用gzip压缩，在名字中包含软件包的版本（例如foo-0.2.1.tar.gz）。它们应该解开到单个目录中，目录应该以软件包及其版本命名，比如foo-0.2.1。
- 国际化是由GNU gettext软件包提供的。将gettext软件包随应用程序提供给用户，这样用户没有gettext也能够实现国际化。

下面是创建Gtk+/Gnome应用程序源代码树框架的重要步骤：

1) 创建一个顶级目录，用以容纳应用程序的所有组件，包括编译文件、文档以及翻译文件。

2) 通常在顶级目录下创建一个src子目录，将所有的源代码放在该目录下，并与其他文件分开。

3) 在顶级目录下，创建AUTHORS、NEWS、COPYING和README文件。还可以创建一个空的ChangeLog文件。

4) 写一个configure.in文件；configure.in文件的主要作用是决定使用什么样的编译器、编译标志以及链接标志。configure.in还可以使用#define符号反映当前平台的特征；它把这些优先放在自动生成的config.h文件里。

5) 写一个acconfig.h文件。它是config.h.in文件要使用的模板文件。这个文件应该撤销每个可能在config.h中定义了的符号以避免重复定义（一般在config.h中用#define定义，用#undef撤销定义）。autoheader程序基于acconfig.h创建config.h.in文件，autoconf程序创建config.h文件。autoheader是autoconf软件包中的实用程序。

6) 创建一个空的stamp.h.in文件。在configure.in中的AM_CONFIG_HEADER宏会用到它。

7) 在顶级目录下，写一个Makefile.am文件，在其中列出每个包含源代码的子目录；在每个子目录中也写一个Makefile.am文件。

8) 运行gettext软件包中的gettextize程序。这样可以创建intl和po目录，这是软件国际化所需要的。在intl目录中包含GNU gettext源代码。如果编译程序的用户没有gettext，它们可以在执行configure脚本时传一个--with-included-gettext参数，让configure在intl目录下自动编译一个gettext的静态版本。在po容纳了翻译文件后，gettextize也会创建一个称为po/Makefile.in.in的文件，用于编译翻译文件。

9) 创建一个po/POTFILES.in文件，在其中列出应该扫描字符串以便翻译的源文件。最初的POTFILES.in文件可以是空的。

10) 从其他的Gnome模块中复制一个autogen.sh文件和它的宏目录。必须根据自己的软件包的名称修改autogen.sh文件。运行autogen.sh文件将调用libtoolize、aclocal、autoheader、automake以及autoconf。

11) autogen.sh用--add-missing参数调用文件automake。这会添加一些文件，比如带有通用安装指导的INSTALL文件。编辑INSTALL，在其中包含任何针对应用程序的安装指南。

autoconf.sh会在每个目录下创建一个 Makefile。

2.11.2 configure.in文件

autoconf处理configure.in文件，生成一个configure脚本。configure是一个可移植的shell脚本，它检查编译环境以决定哪些库可用，所用平台有什么特征，哪些库和头文件已经找到等等。基于这些信息，它修改编译标记，生成 Makefile文件，并/或输出一个包含已定义的预处理符号的config.h文件。configure并不需要运行autoconf，所以在发布应用程序之前生成这个文件，这样，用户就不必有 autoconf软件包。

眼前的任务就是写一个configure.in文件。文件基本上是一系列的 m4宏，根据传递给它们的参数，这些宏扩展为 shell脚本代码段。还可以手工书写 shell代码。要真正理解怎样写configure.in文件要求有一些 m4的知识以及一些 Bourne shell的知识。幸运的是，有省事的方法；可以找一个已有的 configure.in文件，然后修改它以适应你的应用程序。还有一个很全面的autoconf 手册，里面介绍了很多随 autoconf发布的预先写好的宏。

Gtk+和Gnome的开发者已经进一步简化了这些工作，提供了一些宏用于在用户的系统中定位Gtk+和Gnome。

下面是一个简单的configure.in文件，来自于Gnome版的“Hello, World”：

```
AC_INIT(src/hello.c)
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(GnomeHello, 0.1)
AM_MAINTAINER_MODE
AM_ACLOCAL_INCLUDE(macros)
GNOME_INIT
AC_PROG_CC
AC_ISC_POSIX
AC_HEADER_STDC
AC_ARG_PROGRAM
AM_PROG_LIBTOOL
GNOME_COMPILE_WARNINGS
ALL_LINGUAS="de es fr no ru sv fi"
AM_GNU_GETTEXT
AC_SUBST(CFLAGS)
AC_SUBST(CPPFLAGS)
AC_SUBST(LDFLAGS)
AC_OUTPUT([
Makefile
macros/Makefile
src/Makefile
intl/Makefile
po/Makefile.in
pixmaps/Makefile
doc/Makefile
doc/C/Makefile
doc/es/Makefile
])
```

上面以 AC开头的宏来自 autoconf，以 AM开头的宏来自 automake。要从 autoconf或

automake中寻求帮助，这一点很有用。以GNOME开头的宏来自于Gnomemacros目录。这些宏都是用m4宏语言写的。如果将autoconf和automake安装在/usr目录下，autoconf和automake中的标准宏一般放在/usr/share/aclocal目录下。

- AC_INIT总是configure.in中的第一个宏。它扩展为许多可由其他 configure脚本共享的模板文件代码。这些代码解析传到 configure中的命令行参数。这个宏的一个参数是一个文件名，这个文件应该在源代码目录中，它用于健全性检查，以保证 configure脚本已正确定位源文件目录。
- AM_CONFIG_HEADER指定了要创建的头文件，差不多总是 config.h。创建的头文件包含由configure定义的C预处理符号。最低限度应该定义 PACKAGE和VERSION符号，这样可以将应用程序名称和版本传送到代码中，而无须对它们硬编码（非公用的源文件应该包含 config.h(#include <config.h>)以利用这些定义。然而，不要将 config.h文件安装到系统中，因为它有可能与其他的软件包冲突)。
- AM_INIT_AUTOMAKE初始化 automake。传到这个宏里的参数是要编译的应用程序的名称和版本号(这些参数成为 config.h中定义的 PACKAGE和VERSION值)。
- AM_MAINTAINER_MODE关闭缺省时仅供程序维护者使用的 makefile目标，并修改以使configure能理解--enable-maintainer-mode选项。--enable-maintainer-mode将maintainer-only目标重新打开。仅供维护者使用的 makefile目标允许最终用户清除自动生成的文件，比如configure，这意味着要修复编译故障，必须安装有 autoconf 和automake软件。注意，因为 autogen.sh脚本主要是给开发人员用的， autogen.sh会自动传递一个 --enable-maintainer-mode选项给configure。
- AM_ACLOCAL_INCLUDE指定一个附加的目录，用于搜索 m4宏。在这里，它指定为 macros子目录。在这个目录中应该有 Gnome宏的拷贝。
- GNOME_INIT给configure添加一个与 Gnome相关的命令行参数个数，并为 Gnome程序定义一些makefile变量，这些变量中包含了必要的预处理程序和链接程序标志。这些标志是由 gnome-config脚本取得的。安装 gnome-libs时会安装 gnome-config脚本。
- AC_PROG_CC定位C编译器。
- AC_ISC_POSIX 添加一些在某些平台上实现 POSIX兼容需要的标志。
- AC_HEADER_STDC检查当前平台上是否有标准的 ANSI头文件，如果有，则定义 STDC_HEADERS。
- AC_ARG_PROGRAM添加一些选项到configure中，让用户能够修改安装程序的名称（如果在用户系统上碰巧有一个与要安装的程序名称相同的程序，这是很有用的）。
- AM_PROG_LIBTOOL是由automake用来设置 libtool的用途的。只在计划编译共享库或动态可加载模块时才需要设置这个值。
- GNOME_COMPILE_WARNINGS给gcc命令行添加许多警告选项，但是在其他绝大多数的编译器上什么也不做。
- ALL_LINGUAS= “ es ” 不是一个宏，只是一句 shell代码。它包含一个由空格分隔的语言种类缩写表，对应于 po子目录下的 .po文件。 .po文件包含翻译成其他语言的文本，所以ALL_LINGUAS应该列出程序已经被翻译成的所有语言。
- AM_GNU_GETTEXT由automake使用，但是这个宏会随 gettext软件包发布。它让

automake执行一些与国际化相关的任务。

- AC_SUBST输出一个变量到由configure生成的文件中。具体内容将在后面说明。
- AC_OUTPUT列出由configure脚本创建的文件。这些文件都是由带.in后缀的同名文件生成的。例如，src/Makefile是由src/Makefile.in生成的，config.h是由config.h.in生成的。

在执行AC_OUTPUT宏时，configure脚本处理包含有两个@符号标志的变量（例如@PACKAGE@）的文件。只有用AC_SUBST输出了变量，它才能识别这些变量（许多在上面讨论过的预先写好的宏都用AC_SUBST定义变量）。这些特征用于将一个Makefile.in文件转换成一个Makefile文件。典型情况下，Makefile.in是由automake从Makefile.am生成的（不过，你可以只用autoconf，而不用automake，自己编写一个Makefile.in）。

2.11.3 Makefile.am文件

automake处理Makefile.am，生成一个符合标准的Makefile.in文件。automake会做很多工作：例如，它维护源文件之间的依赖关系；生成所有的标准目标，比如install和clean；它还生成更复杂的目标：如果Makefile.am是正确的，简单输入make dist就会创建一个标准的.tar.gz文件。

一般情况是在最上层目录下写一个Makefile.am，然后在每一个子目录下分别写一个Makefile.am文件。automake会从最上层开始递归处理各个Makefile.am，然后生成一个Makefile.in。

在最上层目录的Makefile.am通常都很简单，下面是一个例子：

```
SUBDIRS = macros po intl src pixmaps doc
EXTRA_DIST = \
    gnome-hello.desktop
Applicationsdir = $(datadir)/gnome/apps/Applications
Applications_DATA = gnome-hello.desktop
```

上面程序的第一行通知automake在给定的子目录中递归查找Makefile.am文件。在src子目录的Makefile.am是这样的：

```
INCLUDES = -I$(top_srcdir) -I$(includedir) $(GNOME_INCLUDEDIR) \
    -DG_LOG_DOMAIN=\"GnomeHello\"
-DGNOMELOCALEDIR=\"\"$(datadir)/locale\" \
-I../intl -I$(top_srcdir)/intl
bin_PROGRAMS = gnome-hello
gnome_hello_SOURCES = \
    app.c \
    hello.c \
    menus.c \
app.h \
hello.h \
menus.h
gnome_hello_LDADD = $(GNOMEUI_LIBS) $(GNOME_LIBDIR) $(INTLLIBS)
```

automake能够理解许多“不可思议的变量”，并用这些变量创建Makefile.in文件。在上面的小例子中，用到了下面的变量：

- INCLUDES指定了在编译阶段（与连接阶段相对）中传递给C编译器的标志。这一行用到的变量来自于2.11.2节中的configure.in文件。

- `bin_PROGRAMS`列出了要编译的程序。
- `hello_SOURCES`列出了要编译和连接的文件，这些文件是依赖生成 `hello`程序的。程序名必须列在`bin_PROGRAMS`中。在这个变量中的所有文件都被自动包含在发布包中。
- `hello_LDADD`列出了要传递给连接程序的标志。在这个例子中是由 `configure`决定的 Gnome库标志。
- `INCLUDES`行中有几个在所有的 Gnome程序中都应该用到的元素。应该定义 `G_LOG_DOMAIN`，来自与校验和断言代码中的错误信息会报告这个值，这样就能够判定错误是发生在什么地方(在代码中，还是在一个库中)。GNOMELOCALEDIR用于定位翻译文件。`intl`目录被添加到了头文件的搜索路径，这样应用程序就能够找到 `intl`头文件。

在`Makefile.am`中还可以做很多复杂的事，特别是，可以添加两端带有 `@`符号的、能带入到`configure`脚本中的变量。可以有条件地包含基于 `configure`校验的`Makefile`文件中的一部分，还可以建立库。`automake`的手册介绍了细节内容。

表2-1概括了由 `automake`生成的最常见的 `make`目标。当然，缺省的 `make`目标是`all`，它编译整个程序。GNU代码标准(http://www.gnu.org/prep/standards_toc.html)中有这些 `make`目标和 GNU `Makefile`文件的详细信息。

表2-1 make目标

标准make目标	目标介绍
<code>Dist</code>	建立一个用于发布的tar压缩文件(.tar.gz)
<code>Distcheck</code>	建立一个用于发布的压缩文件，然后尝试编译
<code>Clean</code>	删除编译结果(目标文件和可执行文件)，但是也许不会删除一些由机器生成的文件
<code>Install</code>	如果需要，创建安装目录，将软件复制到目录里
<code>Uninstall</code>	反安装(删除已经安装文件)
<code>Distclean</code>	将执行 <code>configure</code> 脚本以及所有 <code>make</code> 过程的效果按相反的过程重做一遍，也就是，将一个tar文件还原为它原来的状态
<code>Mostlyclean</code>	和 <code>clean</code> 差不多，但是将那些极有可能不需重建的，目标文件留下来
<code>Maintainer-clean</code>	比 <code>clean</code> 更彻底，也许会删除一些需要由特殊工具软件重建的文件，比如由机器生成的源代码
<code>TAGS</code>	重建一个tag表，Emacs可以使用它
<code>Check</code>	如果有，则运行一个测试组件

2.11.4 安装支持文件

完整的Gnome应用程序还有许多代码以外的东西。它们有在线帮助(要列在 Gnome的主菜单上)，有界面翻译，还有一个桌面图标。它们也许带一个 `pixmap`以及一个用在“关于”对话框上的徽标、一个用于“向导”的图形或者一个用以帮助用户快速区别菜单项或列表元素的小图标。下面的内容介绍怎样发布这些文件。

1. 安装数据文件：文档和 `pixmap`

文档和 `pixmap`的安装方法是差不多的。`automake`允许你将数据文件安装到任意位置，可以用配置文件中定义的变量决定将它们安装到哪里。

(1) `pixmap`

要从Makefile.am中安装数据文件，只需简单地为安装目标指定一个名字 (pixmap 就不错) 然后为该目录和要安装到目录里的文件分别创建一个变量。例如：

```
EXTRA_DIST = gnome-hello-logo.png
pixmapdir = $(datadir)/pixmaps
pixmap_DATA = gnome-hello-logo.png
fill
```

“ pixmap ” 字符串将 pixmap_DATA 变量和 pixmapdir 变量连接起来。 automake 解释 _DATA 前缀，并在 Makefile.in 中生成适当的安装规则。这个 Makefile.am 片断将 gnome-hello-logo.png 文件安装到 \$(datadir)/pixmaps 目录下，\$(datadir) 是由 configure 分配的变量。典型情况下，\$(datadir) 是 /usr/local/share (更精确的说，是 \$(prefix)/share)，这是独立于体系结构的数据文件 (也就是，几个具有不同二进制文件格式的系统共享的文件) 的标准位置。

Gnome 的 pixmap 图片的标准位置是 \$(datadir)/pixmaps，所以在例子中我们这样用。Gnome 项目鼓励在所有的 pixmap 图片中使用 PNG 格式，这个格式是 gdk_imlib (Gnome 图象加载库) 支持的。它的文件尺寸小，速度快，也不存在专利问题。

(2) 文档

安装文档使用同样的原则，不过稍有一点复杂。 Gnome 文档通常是用 DocBook 写的。DocBook 是一个 SGML DTD (Document Type Definition，文档类型定义)，就像 HTML 一样。然而，DocBook 的文档标签是为技术文档设计的。用 DocBook 写的文档可以转换为其他格式，包括 PostScript 和 HTML。依照标准，应该安装 HTML 格式的文档，用户就可以用 Web 浏览器或 Gnome 帮助浏览器阅读文档。

Gnome 库和帮助浏览器能理解一个名为 topic.dat 的文件，这个文件只是一个含有相应 URL 的帮助主题列表。它起应用程序帮助主题索引的作用。下面是一个例子，只有两条：

```
gnome-hello.html      GnomeHello manual
advanced.html         Advanced Topics
```

URL 路径相对于所安装的帮助文件的目录。

应该预先考虑文档可能会翻译为其他语言。最好为每一个地区建立一个子目录，例如，缺省地区 (C) 或 es (西班牙语)。使用这种方法翻译程序不会引起混乱。一般将 Gnome 帮助安装在以地区开头的目录下，这种做法用其他观点来看也是很方便的。文档目录看起来也许和 GnomeHello 示例程序差不多：

```
doc/
  Makefile.am
C/
  Makefile.am
  gnome-hello.sgml
  topic.dat
es/
  Makefile.am
  gnome-hello.sgml
  topic.dat
```

下面是 doc/C/Makefile.am：

```
gnome_hello_helpdir = $(datadir)/gnome/help/gnome-hello/C
```



```

gnome_hello_help_DATA = \
    gnome-hello.html \
    topic.dat
SGML_FILES = \
    gnome-hello.sgml
# files that aren't in a binary/data/library target have to be listed here
# to be included in the tarball when 'make dist'
EXTRA_DIST = \
    topic.dat \
    $(SGML_FILES)
## The - before the command means to ignore it if it fails. That way
## people can still build the software without the docbook tools
all:
gnome-hello.html: gnome-hello/gnome-hello.html
-cp gnome-hello/gnome-hello.html .
gnome-hello/gnome-hello.html: $(SGML_FILES)
-db2html gnome-hello.sgml
## when we make dist, we include the generated HTML so people don
## have to have the docbook tools
dist-hook:
mkdir $(distdir)/gnome-hello
-cp gnome-hello/*.html gnome-hello/*.css $(distdir)/gnome-hello
-cp gnome-hello.html $(distdir)
install-data-local: gnome-hello.html
$(mkinstalldirs) $(gnome_hello_helpdir)/images
-for file in $(srcdir)/gnome-hello/*.html $(srcdir)/gnome-hello/*.css; do \
basefile=`basename $$file`; \
$(INSTALL_DATA) $(srcdir)/$$file $(gnome_hello_helpdir)/$$basefile; \
done
gnome-hello.ps: gnome-hello.sgml
-db2ps $<
gnome-hello.rtf: gnome-hello.sgml
-db2rtf $<

```

需要特别注意的是生成的 HTML 文件的安装目录：\$(datadir)/gnome/help/gnome-hello/C。Gnome 库在这里查找帮助文件。每个应用程序的帮助都放在 \$(datadir)/gnome/help 下的它自己的目录下。每个地区的文档都安装在应用程序目录的对应于地区的子目录下。

2. .desktop 入口

Gnome 程序带有一个 .desktop 入口，它是一个以 desktop 为后缀的文本文件，描述应用程序应该放在 Gnome 主菜单的什么位置。安装一个 .desktop 入口文件可以让应用程序显示在 Gnome 面板菜单（主菜单）中。下面是 gnome-hello.desktop 文件：

```

[Desktop Entry]
Name=Gnome Hello
Name[es]=Gnome Hola
Name[fi]=GNOME-hei
Name[no]=Gnome hallo
Name[sv]=Gnome Hej
Comment=Hello World
Comment[es]=Hola Mundo

```

```
Comment[fi]=Hei, maaailma
Comment[sv]=Hej Världen
Comment[no]=Hallo verden
Exec=gnome-hello
Icon=gnome-hello-logo.png
Terminal=0
Type=Application
```

这个文件由键 - 值对组成。Name键指定在缺省地区场合的名称；任何键都可以有一个用于标明地区的字符串，放在一对方括号里面。当登录到 X窗口时，如果指定了不同的地区，系统会根据语言从这里读取相应的字符串。Comment键是一个“工具提示”或“暗示”，用以更详细地描述应用程序。Exec是用来执行程序命令行。Terminal是布尔类型，如果为非0值，程序在一个终端内运行。在这里Type应该是“Application”

安装一个.desktop条目很简单。下面是GnomeHello中的最顶层的Makefile.am文件：

```
SUBDIRS = macros po intl src pixmaps doc
EXTRA_DIST = \
    gnome-hello.desktop
Applicationsdir = $(datadir)/gnome/apps/Applications
Applications_DATA = gnome-hello.desktop
```

注意，在\$(datadir)/gnome/apps/下有一个目录树，可以将应用程序安装到下面子目录所对应的类别中。GnomeHello将自己安装在“Applications”类中，而其他的应用程序也许会选择Games、Graphics、Internet或者其他合适的地方。尽量选择一个已经存在的类别，而不是自己建一个。

Makefile.am中的EXTRA_DIST变量列出了需要包含在发布软件包（压缩的tar文件）中的文件。最重要的文件会自动包含进来，例如，所有作为二进制或库的源文件的文件都会自动包含。然而，automake并不认识.desktop文件，或SGML文档，所有这些文件必须列在EXTRA_DIST中。如果将这些文件留在EXTRA_DIST之外，用make distcheck命令编译发布程序通常会失败。

第二部分 Linux编程常用C语言 函数库及构件库

第3章 glib库简介

glib库是Linux平台下最常用的C语言函数库，它具有很好的可移植性和实用性。glib是Gtk+库和Gnome的基础。glib可以在多个平台下使用，比如Linux、Unix、Windows等。glib为许多标准的、常用的C语言结构提供了相应的替代物。如果有什么东西本书没有介绍到，请参考glib的头文件：glib.h。glib.h中的头文件很容易理解，很多函数从字面上都能猜出它的用处和用法。如果有兴趣，glib的源代码也是非常好的学习材料。

glib的各种实用程序具有一致的接口。它的编码风格是半面向对象，标识符加了一个前缀“g”，这也是一种通行的命名约定。

使用glib库的程序都应该包含glib的头文件glib.h。如果程序已经包含了gtk.h或gnome.h，则不需要再包含glib.h。

3.1 类型定义

glib的类型定义不是使用C的标准类型，它自己有一套类型系统。它们比常用的C语言的类型更丰富，也更安全可靠。引进这套系统是为了多种原因。例如，gint32能保证是32位的整数，一些不是标准C的类型也能保证。有一些仅仅是为了输入方便，比如guint比unsigned更容易输入。还有一些仅仅是为了保持一致的命名规则，比如，gchar和char是完全一样的。

以下是glib基本类型定义：

整数类型：gint8、guint8、gint16、guint16、gint32、guint32、gint64、guint64。其中gint8是8位的整数，guint8是8位的无符号整数，其他依此类推。这些整数类型能够保证大小。不是所有的平台都提供64位整型，如果一个平台有这些，glib会定义G_HAVE_GINT64。

整数类型gshort、glong、gint和short、long、int完全等价。

布尔类型gboolean：它可使代码更易读，因为普通C没有布尔类型。Gboolean可以取两个值：TRUE和FALSE。实际上FALSE定义为0，而TRUE定义为非零值。

字符型gchar和char完全一样，只是为了保持一致的命名。

浮点类型gfloat、gdouble和float、double完全等价。

指针gpointer对应于标准C的void*，但是比void*更方便。

指针gconstpointer对应于标准C的const void*（注意，将const void*定义为const gpointer是行不通的）。

3.2 glib的宏

3.2.1 常用宏

glib定义了一些在C程序中常见的宏，详见下面的列表。TRUE/FALSE/NULL就是

1/0/((void*)0)。MIN()/MAX()返回更小或更大的参数。ABS()返回绝对值。CLAMP(x, low, high)若X在[low, high]范围内,则等于X;如果X小于low,则返回low;如果X大于high,则返回high。

一些常用的宏列表

```
#include <glib.h>
TRUE
FALSE
NULL
MAX(a, b)
MIN(a, b)
ABS(x)
CLAMP(x, low, high)
```

有些宏只有glib拥有,例如在后面要介绍的gpointer-to-gint和gpointer-to-guint。

大多数glib的数据结构都设计成存储一个gpointer。如果想存储指针来动态分配对象,可以这样做。然而,有时还是想存储一系列整数而不想动态地分配它们。虽然C标准不能严格保证,但是在多数glib支持的平台上,在gpointer变量中存储gint或guint仍是可能的。在某些情况下,需要使用中间类型转换。

下面是示例:

```
gint my_int;
gpointer my_pointer;
my_int = 5;
my_pointer = GINT_TO_POINTER(my_int);
printf("We are storing %d\n", GPOINTER_TO_INT(my_pointer));
```

这些宏允许在一个指针中存储一个整数,但在一个整数中存储一个指针是不行的。如果要实现的话,必须在一个长整型中存储指针。

宏列表:在指针中存储整数的宏

```
#include <glib.h>
GINT_TO_POINTER(p)
GPOINTER_TO_INT(p)
GUINT_TO_POINTER(p)
GPOINTER_TO_UINT(p)
```

3.2.2 调试宏

glib提供了一整套宏,在你的代码中使用它们可以强制执行不变式和前置条件。这些宏很稳定,也容易使用,因而Gtk+大量使用它们。定义了G_DISABLE_CHECKS或G_DISABLE_ASSERT之后,编译时它们就会消失,所以在软件代码中使用它们不会有性能损失。大量使用它们能够更快速地发现程序的错误。发现错误后,为确保错误不会在以后的版本中出现,可以添加断言和检查。特别是当编写的代码被其他程序员当作黑盒子使用时,这种检查很有用。用户会立刻知道在调用你的代码时发生了什么错误,而不是猜测你的代码中有什么缺陷。

当然,应该确保代码不是依赖于一些只用于调试的语句才能正常工作。如果一些语句在生成代码时要取消,这些语句不应该有任何副作用。

宏列表:前提条件检查

```
#include <glib.h>
g_return_if_fail(condition)
g_return_val_if_fail(condition, retval)
```

这个宏列表列出了 glib 的预条件检查宏。对 `g_return_if_fail()`，如果条件为假，则打印一个警告信息并且从当前函数立刻返回。`g_return_val_if_fail()` 与前一个宏类似，但是允许返回一个值。毫无疑问，这些宏很有用——如果大量使用它们，特别是结合 Gtk+ 的实时类型检查，会节省大量的查找指针和类型错误的时间。

使用这些函数很简单，下面的例子是 glib 中哈希表的实现：

```
void
g_hash_table_foreach (GHashTable *hash_table,
                     GFunc      func,
                     gpointer    user_data)
{
    GHashNode *node;
    gint i;

    g_return_if_fail (hash_table != NULL);
    g_return_if_fail (func != NULL);

    for (i = 0; i < hash_table->size; i++)
        for (node = hash_table->nodes[i]; node; node = node->next)
            (* func) (node->key, node->value, user_data);
}
```

如果不检查，这个程序把 NULL 作为参数时将导致一个奇怪的错误。库函数的使用者可能要通过调试器找出错误出现在哪里，甚至要到 glib 的源代码中查找代码的错误是什么。使用这种前提条件检查，他们将得到一个很不错的错误信息，告之不允许使用 NULL 参数。

宏列表：断言

```
#include <glib.h>
g_assert(condition)
g_assert_not_reached()
```

glib 也有更传统的断言函数。`g_assert()` 基本上与 `assert()` 一样，但是对 `G_DISABLE_ASSERT` 响应（如果定义了 `G_DISABLE_ASSERT`，则这些语句在编译时不编译进去），以及在所有平台上行为都是一致的。还有一个 `g_assert_not_reached()`，如果执行到这个语句，它会调用 `abort()` 退出程序并且（如果环境支持）转储一个可用于调试的 core 文件。

应该断言用来检查函数或库内部的一致性。`g_return_if_fail()` 确保传递到程序模块的公用接口的值是合法的。也就是说，如果断言失败，将返回一条信息，通常应该在包含断言的模块中查找错误；如果 `g_return_if_fail()` 检查失败，通常要在调用这个模块的代码中查找错误。这也是断言与前提条件检查的区别。

下面 glib 日历计算模块的代码说明了这种差别：

```
GDate*
g_date_new_dmy (GDateDay day, GDateMonth m, GDateYear y)
{
    GDate *d;
    g_return_val_if_fail (g_date_valid_dmy (day, m, y), NULL);
    d = g_new (GDate, 1);
```

```
d->julian = FALSE;
d->dmy    = TRUE;

d->month  = m;
d->day    = day;
d->year   = y;

g_assert (g_date_valid (d));

return d;
}
```

开始的预条件检查确保用户传递合理的年月日值；结尾的断言确保 glib构造一个健全的对象，输出健全的值。

断言函数 `g_assert_not_reached()` 用来标识“不可能”的情况，通常用来检测不能处理的所有可能枚举值的 switch 语句：

```
switch (val)
{
    case FOO_ONE:
        break;
    case FOO_TWO:
        break;
    default:
        /* 无效枚举值 */
        g_assert_not_reached();
        break;
}
```

所有调试宏使用 glib 的 `g_log()` 输出警告信息，`g_log()` 的警告信息包含发生错误的应用程序或库函数名字，并且还可以使用一个替代的警告打印例程。例如，可以将所有警告信息发送到对话框或 log 文件而不是输出到控制台。

3.3 内存管理

glib 用自己的 `g_` 变体包装了标准的 `malloc()` 和 `free()`，即 `g_malloc()` 和 `g_free()`。它们有以下几个小优点：

- `g_malloc()` 总是返回 `gpointer`，而不是 `char*`，所以不必转换返回值。
- 如果低层的 `malloc()` 失败，`g_malloc()` 将退出程序，所以不必检查返回值是否是 `NULL`。
- `g_malloc()` 对于分配 0 字节返回 `NULL`。
- `g_free()` 忽略任何传递给它的 `NULL` 指针。

除了这些次要的便利，`g_malloc()` 和 `g_free()` 支持各种内存调试和剖析。如果将 `enable-mem-check` 选项传递给 glib 的 `configure` 脚本，在释放同一个指针两次时，`g_free()` 将发出警告。`enable-mem-profile` 选项使代码使用统计来维护内存。调用 `g_mem_profile()` 时，信息会输出到控制台上。最后，还可以定义 `USE_DMALLOC`，GLIB 内存封装函数会使用 `malloc()`。调试宏在某些平台上在 `dmalloc.h` 中定义。

函数列表：glib 内存分配

```
#include <glib.h>
gpointer g_malloc(gulong size)
void g_free(gpointer mem)
gpointer g_realloc(gpointer mem,
                  gulong size)
gpointer g_memdup(gconstpointer mem,
                  guint bytesize)
```

用g_free()和g_malloc(), malloc()和free(), 以及(如果正在使用C++)new 和 delete匹配是很重要的, 否则, 由于这些内存分配函数使用不同内存池 (new/delete调用构造函数和解构函数), 不匹配将会发生很糟糕的事。

另外, g_realloc()和realloc()是等价的。还有一个很方便的函数 g_malloc0(), 它将分配的内存每一位都设置为0; 另一个函数g_memdup()返回一个从mem开始的字节数为bytesize的拷贝。为了与 g_malloc()一致, g_realloc()和g_malloc0()都可以分配 0字节内存。不过, g_memdup()不能这样做。g_malloc0()在分配的原始内存中填充未设置的位, 而不是设置为数值0。偶尔会有人期望得到初始化为 0.0的浮点数组, 但这样是做不到的。

最后, 还有一些指定类型内存分配的宏, 见下面的宏列表。这些宏中的每一个 type参数都是数据类型名, count参数是指分配字节数。这些宏能节省大量的输入和操纵数据类型的时间, 还可以减少错误。它们会自动转换为目标指针类型, 所以试图将分配的内存赋给错误的指针类型, 应该触发一个编译器警告。

宏列表: 内存分配宏

```
#include <glib.h>
g_new(type, count)
g_new0(type, count)
g_renew(type, mem, count)
```

3.4 字符串处理

glib提供了很丰富的字符串处理函数, 其中有一些是 glib独有的, 一些用于解决移植问题。它们都能与glib内存分配例程很好地互操作。

如果需要比gchar *更好的字符串, glib提供了一个GString类型。

函数列表: 字符串操作

```
#include <glib.h>
gint g_snprintf(gchar* buf,
               gulong n,
               const gchar* format,
               ...)
gint g_strcasecmp(const gchar* s1,
                  const gchar* s2)
gint g_strncasecmp(const gchar* s1,
                  const gchar* s2,
                  guint n)
```

上面的函数列表显示了一些 ANSI C函数的glib替代品, 这些函数在ANSI C中是扩展函数, 一般都已经实现, 但不可移植。对普通的 C函数库, 其中的 sprintf()函数有安全漏洞, 容易造成程序崩溃, 而相对安全并得到充分实现的 snprintf()函数一般都是软件供应商的扩展版本。

在含有snprintf()的平台上，g_snprintf()封装了一个本地的snprintf()，并且比原有实现更稳定、安全。以往的snprintf()不保证它所填充的缓冲是以NULL结束的，但g_snprintf()保证了这一点。

g_snprintf函数在buf参数中生成一个最大长度为n的字符串。其中format是格式字符串，后面的“...”是要插入的参数。

g_strcasecmp()和g_strncasecmp()实现两个字符串大小写不敏感的比较，后者可指定需比较的最大长度。strcasecmp()在多个平台上都是可用的，但是有的平台并没有，所以建议使用glib的相应函数。

下面的函数列表中的函数在合适的位置上修改字符串：第一个将字符串转换为小写，第二个将字符串全部转换为大写。g_strreverse()将字符串颠倒过来。g_strchug()和g_strchomp()，前者去掉字符串前的空格，后者去掉结尾的空格。宏g_strstrip()结合这两个函数，删除字符串前后的空格。

函数列表：修改字符串

```
#include <glib.h>
void g_strdown(gchar* string)
void g_strup(gchar* string)
void g_strreverse(gchar* string)
gchar* g_strchug(gchar* string)
gchar* g_strchomp(gchar* string)
```

下面的函数列表显示了几个半标准函数的glib封装。g_strtod类似于strtod()，它把字符串nptr转换为gdouble。*endptr设置为第一个未转换字符，例如，数字后的任何文本。如果转换失败，*endptr设置为nptr值。*endptr可以是NULL，这样函数会忽略这个参数。g_strerror()和g_strsignal()与前面没有“g_”的函数是等价的，但是它们是可移植的，它们返回错误号或警告数的字符串描述。

函数列表：字符串转换

```
#include <glib.h>
gdouble g_strtod(const gchar* nptr,
                 gchar** endptr)
gchar* g_strerror(gint errnum)
gchar* g_strsignal(gint signum)
```

下面的函数列表显示了glib中的字符串分配函数。

g_strdup()和g_strndup()返回一个已分配内存的字符串或字符串前n个字符的拷贝。为与glib内存分配函数一致，如果向函数中传递一个NULL指针，它们返回NULL。

printf()返回带格式的字符串。g_strescape在它的参数前面通过插入另一个“\”，将后面的字符转义，返回被转义的字符串。g_strnfill()根据length参数返回填充fill_char字符的字符串。

g_strdup_printf()值得特别注意，它是处理下面代码更简单的方法：

```
gchar* str = g_malloc(256);
g_snprintf(str, 256, "%d printf-style %s", 1, "format");
```

用下面的代码，不需计算缓冲区的大小：

```
gchar* str = g_strdup_printf("%d printf-style %", 1, "format");
```

函数列表：分配字符串


```
#include <glib.h>
gchar*
g_strdup(const gchar* str)
gchar* g_strdup(const gchar* format,
               guint n)
gchar* g_strdup_printf(const gchar* format,
                      ...)
gchar* g_strdup_vprintf(const gchar* format,
                       va_list args)
gchar* g_strescape(gchar* string)
gchar* g_strnfill(guint length,
                 gchar fill_char)
```

`g_strconcat()` 返回由连接每个参数字符串生成的新字符串，最后一个参数必须是 `NULL`，让 `g_strconcat()` 知道何时结束。`g_strjoin()` 与它类似，但是在每个字符串之间插入由 `separator` 指定的分隔符。如果 `separator` 是 `NULL`，则不会插入分隔符。

下面是glib提供的连接字符串的函数。

函数列表：连接字符串的函数

```
#include <glib.h>
gchar* g_strconcat(const gchar* string1,
                  ...)
gchar* g_strjoin(const gchar* separator,
                 ...)
```

最后，下面的函数列表总结了几个处理以 `NULL` 结束的字符串数组的例程。`g_strsplit()` 在每个分隔符处分割字符串，返回一个新分配的字符串数组。`g_strjoinv()` 用可选的分隔符连接字符串数组，返回一个已分配好的字符串。`g_strfreev()` 释放数组中每个字符串，然后释放数组本身。

函数列表：处理以 `NULL` 结尾的字符串向量

```
#include <glib.h>
gchar** g_strsplit(const gchar* string,
                  const gchar* delimiter,
                  gint max_tokens)
gchar* g_strjoinv(const gchar* separator,
                  gchar** str_array)
void g_strfreev(gchar** str_array)
```

3.5 数据结构

glib实现了许多通用数据结构，如单向链表、双向链表、树和哈希表等。下面的内容介绍glib链表、排序二叉树、N-ARY 树以及哈希表的实现。

3.5.1 链表

glib提供了普通的单向链表和双向链表，分别是 `GSLlist` 和 `GList`。这些是由 `gpointer` 链表实现的，可以使用 `GINT_TO_POINTER` 和 `GPOINTER_TO_INT` 宏在链表中保存整数。`GSLlist` 和 `GList` 有一样的API接口，除了有 `g_list_previous()` 函数外没有 `g_slist_previous()` 函数。本节讨论 `GSLlist` 的所有函数，这些也适用于双向链表。

在 glib实现中, 空链表只是一个 NULL指针。因为它是一个长度为 0 的链表, 所以向链表函数传递 NULL总是安全的。以下是创建链表、添加一个元素的代码:

```
GSLIST* list = NULL;
gchar* element = g_strdup("a string");
list = g_slist_append(list, element);
```

glib的链表明显受Lisp的影响, 因此, 空链表是一个特殊的“空”值。g_slist_prepend()操作很像一个恒定时间的操作: 把新元素添加到链表前面的操作所花的时间都是一样的。

注意, 必须将链表用链表修改函数返回的值替换, 以防链表头发生变化。Glib会处理链表的内存问题, 根据需要释放和分配链表链接。

例如, 以下的代码删除上面添加的元素并清空链表:

```
list = g_slist_remove(list, element);
```

链表list现在是 NULL。当然, 仍需自己释放元素。为了清除整个链表, 可使用 g_slist_free(), 它会快速删除所有的链接。因为 g_slist_free()函数总是将链表置为 NULL, 它不会返回值; 并且, 如果愿意, 可以直接为链表赋值。显然, g_slist_free()只释放链表的单元, 它并不知道怎样操作链表内容。

为了访问链表的元素, 可以直接访问 GSLIST结构:

```
gchar* my_data = list->data;
```

为了遍历整个链表, 可以如下操作:

```
GSLIST* tmp = list;
while (tmp != NULL)
{
    printf("List data: %p\n", tmp->data);
    tmp = g_slist_next(tmp);
}
```

下面的列表显示了用于操作 GSLIST元素的基本函数。对所有这些函数, 必须将函数返回值赋给链表指针, 以防链表头发生变化。注意, glib不存储指向链表尾的指针, 所以前插 (prepend) 操作是一个恒定时间的操作, 而追加 (append)、插入和删除所需时间与链表大小成正比。

这意味着用 g_slist_append()构造一个链表是一个很糟糕的主意。当需要一个特殊顺序的列表项时, 可以先调用 g_slist_prepend()前插数据, 然后调用 g_slist_reverse()将链表颠倒过来。如果预计会频繁向链表中追加列表项, 也要为最后的元素保留一个指针。下面的代码可以用来有效地向链表中添加数据:

```
void
efficient_append(GSLIST** list, GSLIST** list_end, gpointer data)
{
    g_return_if_fail(list != NULL);
    g_return_if_fail(list_end != NULL);
    if (*list == NULL)
    {
        g_assert(*list_end == NULL);
        *list = g_slist_append(*list, data);
        *list_end = *list;
    }
}
```

```

else
{
    *list_end = g_slist_append(*list_end, data)->next;
}
}

```

要使用这个函数，应该在其他地方存储指向链表和链表尾的指针，并将地址传递给 `efficient_append()`：

```

GSList* list = NULL;
GSList* list_end = NULL;
efficient_append(&list, &list_end, g_strdup("Foo"));
efficient_append(&list, &list_end, g_strdup("Bar"));
efficient_append(&list, &list_end, g_strdup("Baz"));

```

当然，应该尽量不使用任何改变链表尾但不更新 `list_end` 的链表函数。

函数列表：改变链表内容

```

#include <glib.h>
/* 向链表最后追加数据，应将修改过的链表赋给链表指针 */
GSList* g_slist_append(GSList* list,
                       gpointer data)
/* 向链表最前面添加数据，应将修改过的链表赋给链表指针 */
GSList* g_slist_prepend(GSList* list,
                       gpointer data)
/* 在链表的position位置向链表插入数据，应将修改过的链表赋给链表指针 */
GSList* g_slist_insert(GSList* list,
                      gpointer data,
                      gint position)
/* 删除链表中的data元素，应将修改过的链表赋给链表指针 */
GSList* g_slist_remove(GSList* list,
                      gpointer data)

```

访问链表元素可以使用下面的函数列表中的函数。这些函数都不改变链表的结构。

`g_slist_foreach()` 对链表的每一项调用 `Gfunc` 函数。`Gfunc` 函数是像下面这样定义的：

```
typedef void (*GFunc)(gpointer data, gpointer user_data);
```

在 `g_slist_foreach()` 中，`Gfunc` 函数会对链表的每个 `list->data` 调用一次，将 `user_data` 传递到 `g_slist_foreach()` 函数中。

例如，有一个字符串链表，并且想创建一个类似的链表，让每个字符串做一些变换。下面是相应的代码，使用了前面例子中的 `efficient_append()` 函数。

```

typedef struct _AppendContext AppendContext;
struct _AppendContext {
    GSList* list;
    GSList* list_end;
    const gchar* append;
};
static void
append_foreach(gpointer data, gpointer user_data)
{
    AppendContext* ac = (AppendContext*) user_data;
    gchar* oldstring = (gchar*) data;

```

```

    efficient_append(&ac->list, &ac->list_end,
                    g_strconcat(oldstring, ac->append, NULL));
}
GSList*
copy_with_append(GSList* list_of_strings, const gchar* append)
{
    AppendContext ac;
    ac.list = NULL;
    ac.list_end = NULL;
    ac.append = append;
    g_slist_foreach(list_of_strings, append_foreach, &ac);
    return ac.list;
}

```

函数列表：访问链表中的数据

```

#include <glib.h>
GSList* g_slist_find(GSList* list,
                    gpointer data)
GSList* g_slist_nth(GSList* list,
                    guint n)
gpointer g_slist_nth_data(GSList* list,
                    guint n)
GSList* g_slist_last(GSList* list)
gint g_slist_index(GSList* list,
                    gpointer data)
void g_slist_foreach(GSList* list,
                    GFunc func,
                    gpointer user_data)

```

还有一些很方便的操纵链表的函数，列在下面的函数列表中。除了 `g_slist_copy()` 函数，所有这些函数都影响相应的链表。也就是，必须将返回值赋给链表或某个变量，就像向链表中添加和删除元素时所做的那样。而 `g_slist_copy()` 返回一个新分配的链表，所以能够继续使用两个链表，最后必须将两个链表都释放。

函数列表：操纵链表

```

#include <glib.h>
/* 返回链表的长度 */
guint g_slist_length(GSList* list)
/* 将list1和list2两个链表连接成一个新链表 */
GSList* g_slist_concat(GSList* list1,
                      GSList* list2)
/*将链表的元素颠倒次序*/
GSList* g_slist_reverse(GSList* list)
/*返回链表list的一个拷贝*/
GSList* g_slist_copy(GSList* list)

```

最后，还有一些用于对链表排序的函数，见下面的函数列表。要使用这些函数，必须写一个比较函数 `GcompareFunc`，就像标准C里面的 `qsort()` 函数一样。在 `glib` 里面，比较函数是这个样子：

```
typedef gint (*GcompareFunc) (gconstpointer a, gconstpointer b);
```

如果 $a < b$ ，函数应该返回一个负值；如果 $a > b$ ，返回一个正值；如果 $a = b$ ，返回0。

一旦有了比较函数，就可以将一个元素插入到一个已经排序的链表中，或者对整个链表排序。链表是按升序排序的。使用 `g_slist_find_custom()` 函数，甚至能够循环使用 `GcompareFunc` 来发现链表元素。注意，在 `glib` 中，`GcompareFunc` 的使用是不一致的。有时 `glib` 需要一个等式判定式，而不是一个 `qsort()` 风格的函数。不过，在链表 API 中，它的用法是一致的。

不要随意对链表排序，滥用它们很快就会变得效率低下。例如，`g_slist_insert_sorted()` 函数将数据插入到链表，同时进行排序，它是一个 $O(n)$ 复杂度的操作。但是如果在一个循环中插入多个元素，则每次插入都会进行一次排序，循环的运行时间就是指数级的。较好的方法是先将元素前插，然后调用 `g_slist_sort()` 函数对链表排序。

函数列表：对链表排序

```
#include <glib.h>
GSList* g_slist_insert_sorted(GSList* list,
                              gpointer data,
                              GCompareFunc func)
GSList* g_slist_sort(GSList* list,
                     GCompareFunc func)
GSList* g_slist_find_custom(GSList* list,
                             gpointer data,
                             GCompareFunc func)
```

3.5.2 树

树是一种非常重要的数据结构。在 `glib` 中有两种不同的树：`GTree` 是基本的平衡二叉树，它将存储按键值排序成对键值；`GNode` 存储任意的树结构数据，比如分析树或分类树。

1. GTree

使用下面的函数创建和销毁一个 `Gtree`。其中 `GCompareFunc` 是类似 `GSList` 的 `qsort()` 风格的比较函数。在这里，用它来比较树的键值。

函数列表：创建和销毁平衡二叉树

```
#include <glib.h>
GTree* g_tree_new(GCompareFunc key_compare_func)
void g_tree_destroy(GTree* tree)
```

操作树中数据的函数列在下面的函数列表中。这些函数可以从字面上理解它们的用处和用法。`g_tree_insert()` 覆盖任何已有值，所以如果存在的值是指向已分配内存区域的唯一指针，使用时要当心。如果 `g_tree_lookup()` 没有找到所需的键，返回 `NULL`，否则返回相应的值。键和值都是 `gpointer` 类型，但是要使用整数，并用 `GPOINTER_TO_INT()` 和 `GPOINTER_TO_UINT()` 宏进行转换。

函数列表：操纵 `Gtree` 数据

```
#include <glib.h>
void g_tree_insert(GTree* tree,
                  gpointer key,
                  gpointer value)
void g_tree_remove(GTree* tree,
                  gpointer key)
gpointer g_tree_lookup(GTree* tree,
```

```
gpointer key)
```

下面的函数可以确定树的大小。

函数列表：获得GTree的大小

```
#include <glib.h>
/*获得树的节点数*/
gint g_tree_nnodes(GTree* tree)
/*获得树的高度*/
gint g_tree_height(GTree* tree)
```

使用g_tree_traverse()函数可以遍历整棵树。要使用它，需要一个 GtraverseFunc遍历函数，它用来给g_tree_traverse()函数传递每一对键值对和数据参数。只要 GTraverseFunc返回FALSE，遍历继续；返回 TRUE时，遍历停止。可以用 GTraverseFunc函数按值搜索整棵树。以下是 GTraverseFunc的定义：

```
typedef gint (*GTraverseFunc)(gpointer key, gpointer value, gpointer data);
```

GTraverseType是枚举型，它有四种可能的值。下面是它们在 Gtree中各自的意思：

- G_IN_ORDER(中序遍历)首先递归左子树节点(通过GCompareFunc比较后,较小的键),然后对当前节点的键值对调用遍历函数,最后递归右子树。这种遍历方法是根据使用 GCompareFunc函数从最小到最大遍历。
- G_PRE_ORDER(先序遍历)对当前节点的键值对调用遍历函数,然后递归左子树,最后递归右子树。
- G_POST_ORDER(后序遍历)先递归左子树,然后递归右子树,最后对当前节点的键值对调用遍历函数。
- G_LEVEL_ORDER(水平遍历)在GTree中不允许使用,只能用在Gnode中。

函数列表：遍历GTree

```
#include <glib.h>
void g_tree_traverse(GTree* tree,
                    GTraverseFunc traverse_func,
                    GTraverseType traverse_type,
                    gpointer data)
```

2. GNode

一个GNode是一棵N维的树，由双链表(父和子链表)实现。这样，大多数链表操作函数在Gnode API中都有对等的函数。可以用多种方式遍历。以下是一个 GNode的声明：

```
typedef struct _GNode GNode;
struct _GNode
{
    gpointer data;
    GNode *next;
    GNode *prev;
    GNode *parent;
    GNode *children;
};
```

有一些用来访问GNode成员的宏，见下面的宏列表。作为一个 Glist，其中的data成员可以直接使用。这些宏分别返回 next、prev和children成员，在将GList解除参照以前，这些宏也检查参数是否为NULL，如果是，则返回NULL。

宏列表：访问GNode成员

```
#include <glib.h>
/*返回GNode的前一个节点*/
g_node_prev_sibling(node)
/*返回GNode的下一个节点*/
g_node_next_sibling(node)
/*返回GNode的第一个子节点*/
g_node_first_child(node)
```

用g_node_new()函数创建一个新节点。g_node_new()创建一个包含数据，并且无子节点、无父节点的Gnode节点。通常仅用g_node_new()创建根节点，还有一些宏可以根据需要自动创建新节点。

函数列表：创建一个GNode

```
#include <glib.h>
GNode* g_node_new(gpointer data)
```

要创建一棵树，可以用下面函数列表中的函数。为方便循环或递归树，每个操作都返回刚刚添加的节点。

函数列表：创建一棵GNode树

```
#include <glib.h>
/*在父节点parent的position处插入节点node*/
GNode* g_node_insert(GNode* parent,
                    gint position,
                    GNode* node)
/*在父节点parent中的sibling节点之前插入节点node*/
GNode* g_node_insert_before(GNode* parent,
                           GNode* sibling,
                           GNode* node)
/*在父节点parent最前面插入节点node*/
GNode* g_node_prepend(GNode* parent,
                     GNode* node)
```

下面的宏列表列出了一些常用的宏，用于实现对 Gnode的操作。g_node_append()和g_node_prepend()类似，其余的宏则带一个 data参数，自动分配节点，并且调用相关的基本操作函数。

宏列表：向Gnode添加、插入数据

```
#include <glib.h>
g_node_append(parent, node)
g_node_insert_data(parent, position, data)
g_node_insert_data_before(parent, sibling, data)
g_node_prepend_data(parent, data)
g_node_append_data(parent, data)
```

有两个函数可以从一棵树中删除一个节点。g_node_destroy()从树中删除一个节点，销毁它以及它的子节点。g_node_unlink()将一个节点删除，并将它转换为一个根节点，也就是，它将一棵子树转换为一棵独立的树。

函数列表：销毁GNode

```
#include <glib.h>
void g_node_destroy(GNode* root)
```

```
void g_node_unlink(GNode* node)
```

下面宏列表中的两个宏用来检查一个节点是否是最顶部的节点或最底部的节点。根节点没有父节点和兄弟节点，叶节点没有子节点。

宏列表：判断Gnode的类型

```
#include <glib.h>
G_NODE_IS_ROOT(node)
G_NODE_IS_LEAF(node)
```

下面函数列表中的函数返回 Gnode 的一些有用信息，包括它的节点数、根节点、深度以及含有特定数据指针的节点。其中的遍历类型 GtraverseType 在 Gtree 中介绍过。下面是在 Gnode 中它的可能取值：

- G_IN_ORDER 先递归节点最左边的子树，并访问节点本身，然后递归节点子树的其他部分。这不是很有用，因为多数情况用于 Gtree 中。
- G_PRE_ORDER 访问当前节点，然后递归每一个子树。
- G_POST_ORDER 按序递归每个子树，然后访问当前节点。
- G_LEVEL_ORDER 首先访问节点本身，然后每个子树，然后子树的子树，然后子树的子树的子树，以次类推。也就是说，它先访问深度为 0 的节点，然后是深度为 1，然后是深度为 2，等等。

GNode 的树遍历函数有一个 GTraverseFlags 参数。这是一个位域，用来改变遍历的种类。当前仅有三个标志——只访问叶节点，非叶节点，或者所有节点：

- G_TRAVERSE_LEAFS 指仅遍历叶节点。
- G_TRAVERSE_NON_LEAFS 指仅遍历非叶节点。
- G_TRAVERSE_ALL 只是指(G_TRAVERSE_LEAFS | G_TRAVERSE_NON_LEAFS)快捷方式。

函数列表：取得GNode属性

```
#include <glib.h>
guint g_node_n_nodes(GNode* root,
                     GTraverseFlags flags)
GNode* g_node_get_root(GNode* node)
Gboolean g_node_is_ancestor(GNode* node,
                           GNode* descendant)
Guint g_node_depth(GNode* node)
GNode* g_node_find(GNode* root,
                  GTraverseType order,
                  GTraverseFlags flags,
                  gpointer data)
```

其他GNode函数都很简单，它们大多数是对树的节点表进行操作，见下面的函数列表。

GNode有两个独有的函数类型定义：

```
typedef gboolean (*GNodeTraverseFunc) (GNode* node, gpointer data);
typedef void (*GNodeForeachFunc) (GNode* node, gpointer data);
```

这些函数调用以要访问的节点指针以及用户数据作为参数。 GNodeTraverseFunc 返回 TRUE，停止任何正在进行的遍历，这样就能将 GnodeTraverseFunc 与 g_node_traverse() 结合起来按值搜索树。

函数列表：访问GNode

```
#include <glib.h>
/*对Gnode进行遍历*/
void g_node_traverse(GNode* root,
                    GTraverseType order,
                    GTraverseFlags flags,
                    gint max_depth,
                    GNodeTraverseFunc func,
                    gpointer data)

/*返回GNode的最大高度*/
guint g_node_max_height(GNode* root)
/*对Gnode的每个子节点调用一次func函数*/
void g_node_children_foreach(GNode* node,
                            GTraverseFlags flags,
                            GNodeForeachFunc func,
                            gpointer data)

/*颠倒node的子节点顺序*/
void g_node_reverse_children(GNode* node)
/*返回节点node的子节点个数*/
guint g_node_n_children(GNode* node)
/*返回node的第n个子节点*/
GNode* g_node_nth_child(GNode* node,
                       guint n)
/*返回node的最后一个子节点*/
GNode* g_node_last_child(GNode* node)
/*在node中查找值为date的节点*/
GNode* g_node_find_child(GNode* node,
                        GTraverseFlags flags,
                        gpointer data)
/*返回子节点child在node中的位置*/
gint g_node_child_position(GNode* node,
                          GNode* child)
/*返回数据data在node中的索引号*/
gint g_node_child_index(GNode* node,
                      gpointer data)
/*以子节点形式返回node的第一个兄弟节点*/
GNode* g_node_first_sibling(GNode* node)
/*以子节点形式返回node的第一个兄弟节点*/
GNode* g_node_last_sibling(GNode* node)
```

3.5.3 哈希表

GHashTable是一个简单的哈希表实现，提供一个带有连续时间查寻的关联数组。要使用哈希表，必须提供一个GhashFunc函数，当向它传递一个哈希值时，会返回正整数：

```
typedef guint (*GHashFunc) (gconstpointer key);
```

返回的每个 guint 数值(表字节数的模数)对应于一个哈希表中的一个“存取窗口”或者“哈希表元”。GHashTable通过在每个“存取窗口”中存储一个键值对的链表来处理冲突。因而，GhashFunc函数返回的无符号整数值必须在可能取值中尽可能平均地分配，否则哈希表将

退化为一个链表。GHashFunc也必须快，因为每次查找都要用到它。

除了GhashFunc，还需要一个GcompareFunc比较函数用来测试关键字是否相等。不过，虽然GCompareFunc函数原型是一样的，但它在GHashTable中的用法和在GSList、Gtree中的用法不一样。在GHashTable中可以将GcompareFunc看作是等式操作符，如果参数是相等的，则返回TRUE。当哈希冲突导致在相同的“哈希表元”中有多个关键字 - 值对时，键比较函数用来找到正确的键值对。

使用下面函数列表中的函数创建和销毁一个GHashTable。注意，glib并不知道怎样销毁哈希表中保存的数据，它只销毁表本身。

函数列表：GHashTable

```
#include <glib.h>
GHashTable* g_hash_table_new(GHashFunc hash_func,
                             GCompareFunc key_compare_func)
void g_hash_table_destroy(GHashTable* hash_table)
```

哈希表和比较函数支持最常用的几种键：整数、指针和字符串。这些都列在下面的函数列表中。针对整数的函数接收一个指向 gint 类型的指针，而不是 gint 整数值。如果将 NULL 作为哈希函数的参数传递给 g_hash_table_new()，缺省情况下会使用 g_direct_hash() 函数。如果给键比较函数传递 NULL 参数，那么会使用简单的指针比较函数（等同于 g_direct_equal()，但是没有函数调用）。

函数列表：哈希表/比较函数

```
#include <glib.h>
guint g_int_hash(gconstpointer v)
gint g_int_equal(gconstpointer v1,
                 gconstpointer v2)
guint g_direct_hash(gconstpointer v)
gint g_direct_equal(gconstpointer v1,
                   gconstpointer v2)
guint g_str_hash(gconstpointer v)

gint g_str_equal(gconstpointer v1,
                 gconstpointer v2)
```

操纵哈希表很简单。插入函数不复制键或值，只是将给定的键值准确插入到哈希表，会覆盖任何已存在的具有相同键的键值对（记住，“相同”是由哈希表和比较函数决定的）。如果这样做有问题，在插入前必须查找或删除哈希表的键或值。如果动态分配键或值，需要特别注意。

如果 g_hash_table_lookup() 发现了与键相关联的值，返回这个值，否则，返回 NULL。但有时不能这么做。例如，NULL 可能本身就是一个有效的值。如果使用字符串，特别是动态分配字符串作为键，知道表里的一个键或许并不够，或许想检索出哈希表用来代表“foo”键的确切的 gchar* 值。在这种情况下，可以使用 g_hash_table_lookup_extended()。如果检索成功，g_hash_table_lookup_extended() 返回 TRUE；如果返回 TRUE，则将它发现的键 - 值对放在给定的位置。

函数列表：处理GHashTable

```
#include <glib.h>
```

```

void g_hash_table_insert(GHashTable* hash_table,
                        gpointer key,
                        gpointer value)
void g_hash_table_remove(GHashTable * hash_table,
                        gconstpointer key)
gpointer g_hash_table_lookup(GHashTable * hash_table,
                        gconstpointer key)
gboolean g_hash_table_lookup_extended(GHashTable* hash_table,
                        gconstpointer lookup_key,
                        gpointer* orig_key,
                        gpointer* value)

```

GHashTable 保存一个内部数组，它的大小是质数。它保存存储在表中的键 - 值对数的合计。如果每个有用的“哈希表元”中的键值对平均个数降到 0.3 以下，数组会变小；如果在 3 以上，数组会变大以便减少冲突。不论何时从表中插入或删除键值对，数组都会自动调整大小。这确保了哈希表的内存使用是最优化的。然而，如果正在做大量的插入或删除，会反复重建哈希表，这会急剧降低效率。为了解决这个问题，哈希表可以被“冻结”，即临时禁止调整数组大小。当添加和删除条目已经完成时，简单地“解冻”表，这时会进行一次优化计算。注意，如果添加大量数据，由于哈希冲突，“冻结”的表会“死”掉。在做任何查找以前将表“解冻”就会一切正常。

函数列表：冻结和解冻 GHashTable

```

#include <glib.h>
/**冻结哈希表/
void g_hash_table_freeze(GHashTable* hash_table)
/**将哈希表解冻*/
void g_hash_table_thaw(GHashTable* hash_table)

```

3.6 GString

除了使用 `gchar *` 进行字符串处理以外，Glib 还定义了一种新的数据类型：GString。它类似于标准 C 的字符串类型，但是 GString 能够自动增长。它的字符串数据是以 NULL 结尾的。这些特性可以防止程序中的缓冲溢出。这是一种非常重要的特性。下面是 GString 的定义：

```

struct GString
{
    gchar *str; /* Points to the stringcurrent \0-terminated value. */
    gint len; /* Current length */
};

```

用下面的函数创建新的 GString 变量：

```
GString *g_string_new( gchar *init );
```

这个函数创建一个 GString，将字符串值 `init` 复制到 GString 中，返回一个指向它的指针。如果 `init` 参数是 NULL，创建一个空 GString。

```

void g_string_free( GString *string,
                    gint      free_segment );

```

这个函数释放 `string` 所占据的内存。`free_segment` 参数是一个布尔类型变量。如果 `free_segment` 参数是 TRUE，它还释放其中的字符数据。

```
GString *g_string_assign( GString      *lval,  
                          const gchar *rval );
```

这个函数将字符从 `rval` 复制到 `lval`，销毁 `lval` 的原有内容。注意，如有必要，`lval` 会被加长以容纳字符串的内容。这一点和标准的字符串复制函数 `strcpy()` 相同。

下面的函数的意义都是显而易见的。其中以 `_c` 结尾的函数接受一个字符，而不是字符串。截取 `string` 字符串，生成一个长度为 `len` 的子串：

```
GString *g_string_truncate( GString *string,  
                           gint      len );
```

将字符串 `val` 追加在 `string` 后面，返回一个新字符串：

```
GString *g_string_append( GString *string,  
                          gchar     *val );
```

将字符 `c` 追加到 `string` 后面，返回一个新的字符串：

```
GString *g_string_append_c( GString *string,  
                            gchar     c );
```

将字符串 `val` 插入到 `string` 前面，生成一个新字符串：

```
GString *g_string_prepend( GString *string,  
                           gchar     *val );
```

将字符 `c` 插入到 `string` 前面，生成一个新字符串：

```
GString *g_string_prepend_c( GString *string,  
                             gchar     c );
```

将一个格式化的字符串写到 `string` 中，类似于标准的 `sprintf` 函数：

```
void g_string_sprintf( GString *string,  
                      gchar     *fmt,  
                      ... );
```

将一个格式化字符串追加到 `string` 后面，与上一个函数略有不同：

```
void g_string_sprintfa ( GString *string,  
                        gchar     *fmt,  
                        ... );
```

3.7 计时器函数

计时器函数可以用于为操作计时（例如，记录某项操作用了多长时间）。使用它的第一步是用 `g_timer_new()` 函数创建一个计时器，然后使用 `g_timer_start()` 函数开始对操作计时，使用 `g_timer_stop()` 函数停止对操作计时，用 `g_timer_elapsed()` 函数判定计时器的运行时间。

创建一个新的计时器：

```
GTimer *g_timer_new( void );
```

销毁计时器：

```
void g_timer_destroy( GTimer *timer );
```

开始计时：

```
void g_timer_start( GTimer *timer );
```

停止计时：

```
void g_timer_stop( GTimer *timer );
```

计时重新置零：

```
void g_timer_reset( GTimer *timer );
```

获取计时器流逝的时间：

```
gdouble g_timer_elapsed( GTimer *timer,
                          gulong *microseconds );
```

3.8 错误处理函数

```
gchar *g_strerror( gint errnum );
```

返回一条对应于给定错误代码的错误字符串信息，例如“no such process”等。输出结果一般采用下面这种形式：+

程序名：发生错误的函数名：文件或者描述：strerror

下面是一个使用g_strerror函数的例子：

```
g_print("hello_world:open:%s:%s\n", filename, g_strerror(errno));
void g_error( gchar *format, ... );
```

打印一条错误信息。格式与printf函数类似，但是它在信息前面添加“** ERROR **:”，然后退出程序。它只用于致命错误。

```
void g_warning( gchar *format, ... );
```

与上面的函数类似，在信息前面添加“** WARNING **:”，不退出应用程序。它可以用于不太严重的错误。

```
void g_message( gchar *format, ... );
```

在字符串前添加“message:”，用于显示一条信息。

```
gchar *g_strsignal( gint signum );
```

打印给定信号号码的Linux系统信号的名称。在通用信号处理函数中很有用。

3.9 其他实用函数

glib还提供了一系列实用函数，可以用于获取程序名称、当前目录、临时目录等。这些函数都是在glib.h中定义的。

```
/*返回应用程序的名称*/
```

```
gchar* g_get_prpname (void);
```

```
/*设置应用程序的名称*/
```

```
void g_set_prpname (const gchar *prpname);
```

```
/*返回当前用户的名称*/
```

```
gchar* g_get_user_name (void);
```

```
/*返回用户的真实名称。该名称来自“passwd”文件。返回当前用户的主目录*/
```

```
gchar* g_get_real_name (void);
```

```
/*返回当前使用的临时目录，它按环境变量TMPDIR、TMP and TEMP的顺序查找。如果上面的环境变量都没有定义，返回“/tmp”*/
```

```
gchar* g_get_home_dir (void); gchar* g_get_tmp_dir (void);
```

```
/*返回当前目录。返回的字符串不再需要时应该用 g_free ( )释放*/
gchar* g_get_current_dir (void);
/*获得文件名的不带任何前导目录部分的名称。它返回一个指向给定文件名字符串的指针 */
gchar* g_basename (const gchar *file_name);
/*返回文件名的目录部分。如果文件名不包含目录部分, 返回“.”。返回的字符串不再使用时应该用 g_free
( ) 函数释放*/
gchar* g_dirname (const gchar *file_name);
/*如果给定的file_name是绝对文件名(包含从根目录开始的完整路径, 比如 /usr/local), 返回TRUE*/
gboolean g_path_is_absolute (const gchar *file_name);
/*返回一个指向文件名的根部标志 (“/”) 之后部分的指针。如果文件名 file_name不是一个绝对路径, 返回
NULL*/
gchar* g_path_skip_root (gchar *file_name);
/*指定一个在正常程序终止时要执行的函数 */
void g_atexit (GVoidFunc func);
```

上面介绍的只是 glib库中的一小部分, glib的特性远远不止这些。如果了解其他内容, 请参考 glib.h 文件。这里的绝大多数函数都是简明易懂的。另外, <http://www.gtk.org> 上的 glib 文档也是极好的资源。

如果你需要一些通用的函数, 但 glib 中还没有, 考虑写一个 glib 风格的例程, 将它贡献到 glib 库中! 你自己, 以及全世界的 glib 使用者, 都将因为你的出色工作而受益。

第4章 构件定位

4.1 构件的显现、映射和显示

构件可以按它们是否有 GdKWindow窗口分类。有两种 Gtk+构件，一种有一个相关联的 GdKWindow窗口，另一种没有。大多数构件都有一个相关联的 GdKWindow窗口，构件就绘制在这个窗口上。这里的 GdKWindow窗口和Gtk+里的GtkWindow窗口是不一样的。GdKWindow不是一个用户可见的对象，而是一个 X服务器用于划分屏幕的抽象概念。一个 GdKWindow窗口，对X服务器给出了关于将要显示的图形的结构信息。因为 X窗口系统是网络透明的，有可能X窗口的显示位置和X服务器不在同一台机器上，这样有助于减少网络流量。GtkWindow是一个窗口构件，它是一个用户可见的对象。

还有一些构件，比如说 GtkLabel构件，没有与之相关联的 GdKWindow；它们被称为“无窗口构件”，并且是相对轻量级的。没有相关联窗口的构件绘制在它的父构件的 GdKWindow窗口上。一些操作，例如捕获一个事件，要求有一个 GdKWindow窗口，因此不能在无窗口构件上做这些操作。

构件要经过一系列与它们的 GdKWindow相关的状态：

- 如果一个构件相应的 GdKWindow被创建出来，称为该构件被显现（`realize`）。用 `gtk_widget_realize()`函数显现一个构件，用 `gtk_widget_unrealize()`函数反显现（`unrealized`）构件。因为X窗口必须有一个父窗口，如果一个构件已经显现，它的父窗口也必然已显现。
- 如果在构件的GdKWindow上调用了`gdk_window_show()`函数，称为该构件被映射（`map`）了。这意味着服务器已经要求在屏幕上显示这个构件的 GdKWindow窗口。很明显，GdKWindow窗口必须存在，也就是说，被映射的构件必然已被显现。
- 如果当一个构件的父构件被映射时，它也被映射到屏幕上，这个构件就是可见的。这意味着已经对该构件调用了`gtk_widget_show()`函数。通过调用`gtk_widget_hide()`函数，一个构件可以绘制为不可见的，这或者是取消未决的映射（已经确定了映射的时间，但还未映射），或者反映射该构件（隐藏它的GdKWindow窗口）。因为顶级构件没有父构件，当它们一显示，它们同时就被映射了。

在典型的用户代码中，只需调用 `gtk_widget_show()`函数。这暗含当它的父构件一旦被显现和映射，该构件就被显现和映射。要理解的是：`gtk_widget_show()`函数并不会立即生效，这一点很重要，它仅仅是确定构件被显示出来的时间。也就是说，不用担心显示构件的顺序（不必一定要先显示子构件，再显示父构件）。但是，这时还不能立即访问这个构件的 GdKWindow窗口。有时，又确实需要在映射之前访问构件的 GdKWindow窗口；在这样的情况下，要手工调用 `gtk_widget_realize()`函数来创建这个 GdKWindow。如果机会适当，`gtk_widget_realize()`函数还会显现构件的父构件。使用 `gtk_widget_realize()`函数的情况是不多见的，如果感觉到一定要这么做时，也许是使用了不正确的方法。

上面介绍了创建构件的过程。销毁构件自动地将以上事件的整个次序倒过来，递归取消子构件和构件本身的显现。

正如上面所提到的，对构件调用 `gtk_widget_show()` 函数之后，构件并不一定会显示。只有当构件的所有父构件（直到最高级别的父构件）全部显示之后，它才会显示。因而，一般情况下，应该最后对最高级别的构件 `GtkWindow` 调用 `gtk_widget_show()` 函数。否则，如果先显示高级别的构件，用户可能会看到窗口先出现在屏幕上，然后子构件一个一个显示在屏幕上。用户或许会觉得你的程序不够专业，甚至不正确。

下面是显现、映射和显示构件的相关函数。

函数列表：显示/映射/显现构件

```
#include <gtk/gtkwidget.h>

/* 显现一个构件，创建该构件的GdkWindow*/
void gtk_widget_realize(GtkWidget* widget)

/* 反显现构件，销毁该构件的GdkWindow*/
void gtk_widget_unrealize(GtkWidget* widget)

/*映射构件，构件的GdkWindow显示在窗口上*/
void gtk_widget_map(GtkWidget* widget)

/*反映射构件，隐藏构件的GdkWindow。注意，构件的GdkWindow还存在*/
void gtk_widget_unmap(GtkWidget* widget)

/*显示构件，当构件的父构件（向上递归直到最高级别构件）显示时，
 *构件将显示在屏幕上，*/
void gtk_widget_show(GtkWidget* widget)

/*隐藏构件，构件的GdkWindow依然存在*/
void gtk_widget_hide(GtkWidget* widget)
```

4.2 其他的构件概念

本节介绍了几个其他与 `GtkWidget` 基类相关的概念，其中包括敏感性、焦点以及构件状态。

1. 敏感性

构件可以是敏感或不敏感的，不敏感的构件不能对输入进行响应。一般不敏感的构件是灰色的，不能接收键盘焦点。用 `gtk_widget_set_sensitive()` 函数改变构件的敏感度。

函数列表：改变敏感度

```
#include <gtk/gtkwidget.h>
/*设置构件的敏感性，widget参数是要设置的构件，setting设置为TRUE时，
 *构件是敏感的，setting设置是FALSE时，构件不敏感*/
void gtk_widget_set_sensitive(GtkWidget* widget,
                             gboolean setting)
```

构件缺省是敏感的。只有构件的所有容器是敏感的，构件才能是敏感的。也就是，可以通过设置容器的敏感性来让整个容器内的构件敏感（或不敏感）。构件“真正的”敏感性，包括

它的父构件的状态，可以用 `GTK_WIDGET_IS_SENSITIVE()` 宏测试。构件本身的敏感性，只与构件的父构件的敏感性有关，可以用 `GTK_WIDGET_SENSITIVE()` 宏来查询。

宏列表：敏感性

```
#include <gtk/gtkwidget.h>
GTK_WIDGET_IS_SENSITIVE(widget)
GTK_WIDGET_SENSITIVE(widget)
```

2. 焦点

某个时候，在一个顶级窗口中某个构件可能具有键盘焦点。顶级窗口接收到的任何键盘事件都被发送到这个构件。这一点很重要，因为在键盘上击键应该只有唯一的一种效果，例如，只能更改一个文本输入区域。

大多数构件在具有焦点时，会有一个视觉的指示。当使用缺省的 `Gtk+` 主题 (Theme) 时，典型情况下，有焦点的构件有一个细黑框环绕着。用户可以用方向键或 `Tab` 键在构件之间移动焦点。当用户用鼠标点击构件时，焦点也会移过去。

对键盘导航来说，焦点的概念是很重要的。例如，按下回车键或空格键会“激活”许多具有焦点的构件；可以用 `Tab` 键在按钮之间移动，按下空格键激活这个按钮。

3. 独占

构件能够从其他构件中独占 (`grab`) 鼠标指针和键盘。所谓独占，就是构件是“模态”的，用户只能向这个构件中输入字符，键盘焦点也不能改变到其他构件。独占输入的一个典型理由是：创建一个模态对话框时，如果窗口是独占的，则不能与其他的窗口交互。注意，还有另外一个 `Gdk` 级的“独占”。`Gdk` 键盘和鼠标指针的独占发生在 `X` 服务器范围内，也就是，其他应用程序不能接收到键盘和鼠标事件。构件独占是一个 `Gtk+` 概念，它只独占同一个应用程序中的其他构件的事件。

4. 缺省

每个窗口至多有一个缺省构件。例如，典型情况下，对话框都有一个缺省按钮，当用户按回车键时，相当于点击了这个按钮。

5. 构件状态

构件的状态值决定了它们的外观：

- Normal：就是正常该有的样子。
- Active：例如，按钮正被按下，或检查按钮 (check box) 正被选中。
- Prelight：鼠标指针越过一个构件 (典型情况，按下会有一些效果)。例如，当鼠标越过按钮时，按钮会“高亮显示”。
- Selected：构件是在一个列表中，或者是在其他类似状态，当前它是被选中的。
- Insensitive：构件是“灰色”的，不活动的，或者不响应。它不会对输入响应。

状态的准确含义及其视觉表达依赖于特定构件以及当前窗口管理器的主题 (Theme)。可以用 `GTK_WIDGET_STATE()` 宏存取构件的状态。这个宏返回以下几种常量之一：

```
GTK_STATE_NORMAL
GTK_STATE_ACTIVE
GTK_STATE_PRELIGHT
GTK_STATE_SELECTED
GTK_STATE_INSENSITIVE.
```

宏列表：状态存取函数

```
#include <gtk/gtkwidget.h>
GTK_WIDGET_STATE(widget)
```

4.3 构件的类型转换

在GTK中，所有构件的存储形式都是 GtkWidget，但是许多函数都需要指向某种构件类型（比如 GtkWidget）的指针作为参数。虽然所有的构件都是从 GtkWidget 派生而来的，但是编译器并不能理解这种派生和继承关系。为此，GTK 引进了一套类型转换系统。这些类型转换都是通过宏实现的。这些宏测试给定数据项的类型转换能力，并实现类型转换。下面几个宏是经常会碰到的：

```
GTK_WIDGET(widget)
GTK_OBJECT(object)
GTK_SIGNAL_FUNC(function)
GTK_CONTAINER(container)
GTK_WINDOW(window)
GTK_BOX(box)
```

所有的构件都是从 Object 基类派生而来的，这意味着可以在任何需要一个 GtkWidget 作为参数的函数中使用构件——用 GTK_OBJECT() 宏将构件转换为指向 GtkWidget 类型的指针就可以了。

例如：

```
gtk_signal_connect( GTK_OBJECT(button), "clicked",
                    GTK_SIGNAL_FUNC(callback_function), callback_data);
```

这里将 “ button ” 转换为一个 GtkWidget 对象，并将回调函数名称转换为指向函数的指针。

许多构件也是容器，它们都是从 GtkWidget 内派生而来。这类构件可以用一个 GTK_CONTAINER 宏将它转换为容器，传递到需要容器指针的函数中。

4.4 组装构件

前面介绍了用 GtkWidget/Gnome 构件编写 Linux 应用程序的编程思想。归纳起来就是 “ 事件驱动 ”。也就是，用 GtkWidget/Gnome 构件创建应用程序界面，然后为构件的信号设置回调函数。当用户对界面进行操作时，会引发各种信号。如果某个信号，比如一个按钮的 “ clicked ” 信号连接了回调函数，就会调用这个回调函数。通过在回调函数里面使用代码控制构件的各种属性来与用户交互，或者对内存变量进行操作，实现程序的各种功能。因而，编程的核心就归结为怎样使用构件创建界面，怎样为构件的信号设置回调函数。

在 GtkWidget 版的 “ Hello World ” 的程序中，我们使用 gtk_container_add() 函数将按钮添加到窗口上。如果界面很复杂，不止一个按钮，怎么办呢？怎样在代码中创建构件，怎样将构件在窗口上定位呢？GTK 使用一种称为组装的方法实现了这一点。

GTK 是用 C 语言编写的。虽然没有使用 C++ 这样的面向对象的语言，GTK 实现了自己的具有继承和派生特性的对象系统。在 GtkWidget 构件里面，所有的构件都是从 GtkWidget 对象派生而来的。每种派生构件都继承了父构件接口，同时再实现自己的一些特有功能。这样做的好处是显而易见的，既然新构件的某些功能在已有构件中已经全部具有，为什么不将这个构件直接

拿来用呢？从已有构件派生一个构件，继承旧构件原有的接口，然后再实现那些旧构件不具有的接口和功能就可以创建一个新构件。

在用Gtk+构件创建程序界面时，用容器实现构件的定位。在Gtk+中有两种容器，它们都是抽象的GtkContainer构件的子类。第一种类型的容器构件总是由GtkBin(另一种抽象基类)派生而来。GtkBin的派生类只能容纳一个子构件，它们为其子构件增加一些功能。例如，GtkButton就是一种GtkBin，它让其子构件成为一个可接受点击的按钮。GtkFrame也是一种GtkBin，它在其子构件周围绘制一个边框。同样，GtkWindow让它的子构件显示在一个顶级窗口上。

第二种容器构件通常是直接从GtkContainer派生而来。这些构件可以有多个子构件，它们的作用就是管理布局。“管理布局”意味着这些容器为它们容纳的子构件分配大小尺寸和位置。例如，GtkVBox将它的子构件在一个垂直的栈内排列。GtkTable构件可以让构件在一个表格上根据单元格定位。GtkFixed可以将子构件放在任意坐标位置。GtkPacker允许你做Tk-风格的布局管理。

实际上，Gtk+中的大多数构件都是一个容器。这样也给予我们极大的灵活性。例如，如果我们想要一个带图片的按钮，因为GtkButton是一个容器，只需将一幅图片添加到这个容器里面就可以了。对按钮中是否包含文本，文本和图片之间相对位置等都可以自由设置。使用类似的方法，可以设计出非常复杂，甚至非常古怪的外观布局。

本章介绍第二种容器构件。要生成所需要的布局，并且对构件的尺寸不使用任何硬性编码，需要理解怎样使用这些容器构件。最终的目标之一是避免对窗口尺寸、屏幕尺寸、构件外观、字体等因素做任何假设。如果这些因素发生变化，应用程序应该能够自动适应。

4.4.1 尺寸分配

一个窗口显示在屏幕上，如果用鼠标调整窗口的尺寸，窗口里面的构件会发生什么变化？这依赖于定位构件的容器构件以及构件的定位选项。窗口上的构件可能会按一定的规则改变大小。而这些规则又是通过什么方式实现的呢？这是通过一种称为“请求”和“分配”的协商机制实现的。

1. 请求

构件的请求由宽度和高度组成：构件需要的大小尺寸。它由一个GtkRequisition结构表示：

```
typedef struct _GtkRequisition      GtkRequisition;
struct _GtkRequisition
{
    gint16 width;
    gint16 height;
};
```

不同的构件用不同的方式选择它们所需要的尺寸。例如，GtkLabel构件，请求足够的尺寸用以在标签上显示所有的文本。大多数容器基于它们的子构件的尺寸请求来请求所需尺寸。例如，如果在一个Box里放几个按钮，Box会要求有足够大的空间以容纳所有的按钮。

布局的第一阶段从一个顶级构件，比如说GtkWindow，开始。因为它是一个容器，GtkWindow询问它的子构件的尺寸要求，子构件再询问它的子构件，依此递归。当所有的子构件都被询问过后，GtkWindow最后会从它的子构件得到一个GtkRequisition值。这依赖于它

是如何配置的，GtkWindow也许会或者不会扩展大小以满足所有的尺寸要求。

2. 分配

布局的第二阶段从这一点开始。GtkWindow对可供子构件使用的空间做出一个决定，并向子构件传达这个决定。这就是子构件的尺寸分配，由以下结构表示：

```
typedef struct _GtkAllocation      GtkAllocation;
struct _GtkAllocation
{
    gint16 x;
    gint16 y;
    guint16 width;
    guint16 height;
};
```

width和height元素与GtkRequisition是一样的，它们代表子构件的大小。GtkAllocation结构也包含子构件相对它们父构件的坐标。GtkAllocation由它们的父构件容器分配给子构件。

构件应该尊重传给它们的GtkAllocation值。GtkRequisition仅仅是一个请求，构件必须能够应付任何尺寸。

了解了构件布局的过程，就很容易弄清楚容器在其中扮演的角色。它们的任务就是将每个构件的请求汇总成单个请求，并沿着构件树向上（按层次向上）传递，然后，将它们接收到的大小分配划分给子构件。具体怎样划分依赖于具体的容器。

4.4.2 GtkWindow构件

前面介绍了关于构件的概念、构件的尺寸分配等。创建用户界面的目的就是将各种构件在屏幕上布局，提供一个用户与应用程序交互的接口。大多数情况下，不会将构件直接绘制在屏幕上，而是绘制在一个窗口上。如果窗口的尺寸不变化，而是窗口在屏幕上移动，这些构件在窗口的相对位置一般是固定的，它们随着窗口一起移动。

在Gtk/Gnome应用程序中，GtkWindow构件是最大的容器。GtkWindow是从GtkBin派生而来的，它只能容纳一个子构件。因而，要在其中容纳多个构件，必须使用 GtkBox、GtkTable或者GtkFixed等构件来控制构件布局。

用下面的函数创建新窗口：

```
GtkWidget* gtk_window_new (GtkWindowType type);
```

对于应用程序的窗口，type一般是GTK_WINDOW_TOPLEVEL。

下面的函数用于设置窗口的标题：

```
void gtk_window_set_title (GtkWindow *window,
                           const gchar *title);
```

其中window是要设置标题的窗口，title是一个字符串（窗口标题）。

下面的函数用于设置顶级窗口在处理它的尺寸请求、以及用户调整尺寸时的行为：

```
void          gtk_window_set_policy          (GtkWindow *window,
                                              gint allow_shrink,
                                              gint allow_grow,
                                              gint auto_shrink);
```

注意，不要随意使用这个函数，否则，可能会使窗口的行为变得很古怪。一般只使用下面两种调用方法：


```
gtk_window_set_policy(GTK_WINDOW(window), FALSE, TRUE, FALSE)
```

用户可调整窗口的尺寸。

```
gtk_window_set_policy(GTK_WINDOW(window), FALSE, FALSE, TRUE)
```

窗口的尺寸是由程序控制的，只与窗口的子构件的当前尺寸相匹配。

第一种情况是缺省值，也就是说缺省的窗口就是这样的。

下面的函数用于设置窗口是否可调整尺寸。其中 `setting` 是布尔值，当其值为 `TRUE` 时窗口尺寸可调，为 `FALSE` 时窗口大小不可调整：

```
void gtk_window_set_user_resizeable(GtkWidget* window, gboolean setting);
```

最好只使用下面两种形式设置窗口行为。GTK+ 1.4 可能会用 `gtk_window_set_user_resizeable()` 函数替换 `gtk_window_set_policy()` 函数。

顶级窗口总是改变自己的尺寸以保证它的子构件能够接受到它们的请求值。这意味着如果添加一个子构件，顶级窗口会扩大以容纳它们。不过，如果窗口的尺寸太大，它不会自动缩小以适应子构件的尺寸请求。当 `gtk_window_set_policy()` 函数中的 `auto_shrink` 参数设置为 `TRUE`、子构件的空白区域太多时，窗口将自动缩小以适应子构件的大小。`auto_shrink` 参数通常用在上面提到的两种常见模式中的第二种。也就是说，如果想要让窗口总是根据程序的运行情况自动调整大小，将 `auto_shrink` 设置为 `TRUE`。

注意 如果 `allow_shrink` 和 `allow_grow` 参数都设置为 `FALSE`，`auto_shrink` 参数没有任何作用。

前面提到的两种情况都没有将 `allow_shrink` 参数设置为 `TRUE`。如果 `allow_shrink` 设置为 `TRUE`，用户能够缩小窗口的尺寸，使它的子构件不能接收到全部的尺寸请求。通常这是一个糟糕的主意，因为这样将使大多数构件的外观显示不正确。此外，如果由于某种原因，窗口的尺寸得到重新计算，GTK+ 倾向于重新扩展窗口。因而，`allow_shrink` 参数应该总设置为 `FALSE`。

使用 `allow_shrink` 时，存在的实际问题是一些特殊构件总是需要太多的空间，所以用户不能充分缩小窗口的大小。也许应该对子构件调用 `gtk_widget_set_usize()` 函数，并迫使它的尺寸请求足够大。最好的办法是调用 `gtk_window_set_default_size()` 函数设置构件的缺省尺寸，以便子构件获得比它请求的值更大的分配值。

```
void          gtk_window_set_default_size      (GtkWindow *window,
                                                gint width,
                                                gint height);
```

其中 `width` 和 `height` 是要设置的窗口的缺省宽度和高度。

用下面的构件向窗口中添加子构件：

```
gtk_container_add (GTK_CONTAINER (window), widget);
```

窗口最常用的两个信号是 `delete_event` 和 `destroy`。当使用窗口管理器关闭窗口（点击窗口标题条上的“×”按钮），或者在窗口的某个构件上调用 `gtk_widget_destroy()` 函数时，将引发 `delete_event` 信号。如果在 `delete_event` 信号处理函数中返回 `FALSE`，GTK 将引发 `destroy` 信号。返回 `TRUE` 意味着不需要销毁窗口。对某些窗口，比如对话框，一般不在 `delete_event` 信号中销毁窗口。返回 `FALSE`，窗口会被信号销毁。

一般应该为窗口的 `delete_event` 信号设置一个回调函数，对该信号进行处理。否则，只要

用户点击窗口标题条上的“×”按钮，窗口就会关闭。

4.4.3 GtkBox

有两种GtkBox(组装箱)：GtkHBox(水平组装箱)和GtkVBox(垂直组装箱)。一个GtkBox可以管理一行(GtkHBox)或一列(GtkVBox)构件。对GtkHBox来说，所有的构件都分配了同样的高度，组装箱的作用就是在构件间分配可用空间。GtkHBox还随意用一些可用宽度在构件间预留间隙(称为“间距”)。GtkVBox的作用是一样的，不过是在垂直的方向上(也就是，它分配可用的高度而不是宽度)。GtkBox是一个抽象的基类，GtkVBox和GtkHBox差不多可以完全使用它的接口。组装箱是最有用的容器构件。

用下面函数列表中的构造函数创建新的GtkBox。组装箱构造函数带两个参数。homogeneous参数如果为TRUE(同质)，意味着所有的子构件都被分配同样数量的空间。spacing参数指定每个子构件之间的间距。组装箱创建之后，还有函数可以改变spacing参数值，切换构件的同质属性。

函数列表：GtkHBox构造函数

```
#include <gtk/gtkhbox.h>
GtkWidget* gtk_hbox_new(gboolean homogeneous,
                        gint spacing)
```

函数列表：GtkVBox构造函数

```
#include <gtk/gtkvbox.h>
GtkWidget* gtk_vbox_new(gboolean homogeneous,
                        gint spacing)
```

有两个基本的函数添加子构件到组装箱中，下面的函数列表中已将它们列出。

函数列表：在GtkBox中添加构件

```
#include <gtk/gtkbox.h>
void gtk_box_pack_start(GtkBox* box,
                        GtkWidget* child,
                        gboolean expand,
                        gboolean fill,
                        gint padding)
```

```
void
gtk_box_pack_end(GtkBox* box,
                 GtkWidget* child,
                 gboolean expand,
                 gboolean fill,
                 gint padding)
```

一个组装箱可以包含两套构件。第一套是从组装箱的“头部”(顶部或左边)组装的；第二种是从“尾部”(底部或者右边)开始组装。如果将三个构件从盒子的“头部”开始组装到盒子里，组装的第一个构件会出现在盒子的最上面或者最左边，第二个构件紧接着第一个构件，第三个最接近盒子的中心。如果接着将三个按钮组从组装箱的“尾部”组装到盒子里，第

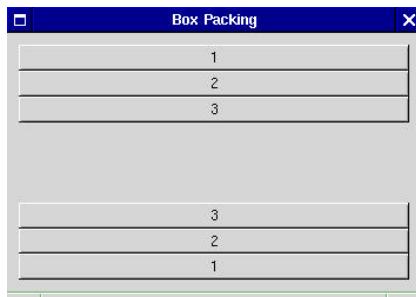


图4-1 组装到GtkVBox中的按钮

一个出现在盒子的最下边或者最右边，第二个紧挨着它，第三个最靠近盒子的中心。所有六个构件都组装之后，从上到下或者从左到右的次序是：1, 2, 3, 3, 2, 1。图4-1显示了这个例子。组装次序只对盒子的每个末端是重要的，也就是，可以使用不同的组装方法，取得相同的结果。

1 GtkBox布局细节

组装过程受三个参数影响，它们对从“头部”和“尾部”组装都是一样的。这些参数的含义很复杂，因为它们与盒子的 `homogeneous` 设置以及每个构件相互作用。

下面介绍 `GtkBox` 怎样计算它的相关“方向”（对 `GtkHBox` 是 `width`，对 `GtkVBox` 是 `height`）上的尺寸请求：

1) 每个子构件的总尺寸请求等于每个子构件的尺寸请求加上两倍用于组装子构件的填充量值。子构件的填充量是在子构件两边的空白空间。简而言之，子构件尺寸 = (子构件的尺寸请求) + 2 × (子构件的填充量)。

2) 如果盒子是同质的，整个盒子的基准尺寸请求等于最大的子构件尺寸请求 (请求数 + 填充量) 乘以子构件的数目。在同质的盒子中，所有的子构件都与最大的子构件一样大的。

3) 如果盒子不是同质的，整个盒子的基准尺寸请求等于每个子构件的尺寸请求 (请求数 + 填充量) 之和。

4) 组装盒范围内的 `spacing` (间距) 设置决定在子构件之间留多大的间距，所以这个值要乘以子构件数目减1，并加到基准尺寸请求中。注意，间距并不属于子构件，它是子构件之间的空白空间，不受 `expand` 和 `fill` 参数的影响。另一方面，`padding` (填充量) 是环绕每个子构件的空间，受子构件的组装参数的影响。

5) 所有的容器都有一个“边框宽度”设置；它会将两倍的边框宽度 (代表两边的边框宽度) 加到尺寸请求中。这样，`GtkBox` 的总的尺寸请求就是：(子构件尺寸之和) + `spacing` × (子构件数 - 1) + 2 × (边框宽度)。

在计算它的尺寸请求，将请求传送到它的父容器之后，`GtkBox` 会接收到它的尺寸分配，并将尺寸按下面的规则在子构件之间分配：

1) 边框宽度和构件间的间距从分配值中减去，剩余的部分是子构件自己的可用空间。这块空间分为两块：子构件实际要求的数量 (子构件的请求值和填充量)，以及“额外空间”。额外空间 = (分配尺寸) - (子构件尺寸之和)。

2) 如果盒子不是同质的，“额外”空间在按 `expand` 参数设置为 `TRUE` 时在子构件之间划分。这些子构件会占满可用空间。如果没有子构件能够展开，额外空间将用于在盒子中央（在从头组装和从尾部组装的构件之间）增加更多的空间。

3) 如果盒子是同质的，额外空间根据需要分配。那些请求较多空间的构件将得到较少的额外空间，让每个构件都占据同样的空间。同质的组装盒忽略 `expand` 参数---额外空间分配给所有的子构件，而不是可扩展的构件。

4) 当构件获得一些额外空间时，有两种可能。在子构件周围增加更多的填充量，或者将子构件本身扩展。`fill` 参数决定哪种可能会发生。如果 `fill` 是 `TRUE`，子构件展开填充空间---也就是，整个空间变成子构件的分配值；如果 `fill` 是 `FALSE`，增加子构件的填充量，填满额外空间，只给子构件分配它请求的空间。注意，如果 `expand` 设置为 `FALSE` 并且盒子也不是同质的，`fill` 没有效果，因为子构件永远也不会接受到额外空间。

相信很少有人能记住这么罗嗦的规则。幸好，Gtk+教程的作者将这么多的设置归纳为5种情形，下面我们一步一步看。

2. 非同质组装盒的组装模式

在非同质的组装盒中有三种组装方法。第一种，将所有的构件用它们的正常尺寸组装到盒子的尾部。这意味着将expand参数设置为FALSE：

```
gtk_box_pack_start(GTK_BOX(box),  
    child,  
    FALSE, FALSE, 0);
```

结果显示在图4-2中。在这里，只有expand参数管用，没有子构件会接受额外空间，所以即使fill设置为TRUE它们也不能充满剩余空间。

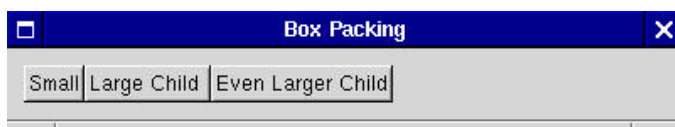


图4-2 非同质，expand = FALSE

第二种方法，可以在GtkBox中扩展构件，让它们保持它们的正常尺寸，如图4-3所示，这意味着将expand参数设置为TRUE：

```
gtk_box_pack_start(GTK_BOX(box),  
    child,  
    TRUE, FALSE, 0);
```

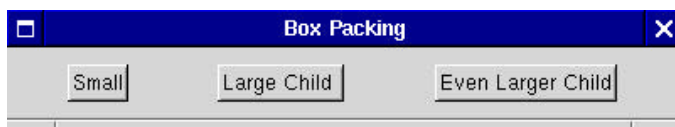


图4-3 非同质，expand = TRUE 和 fill = FALSE

最后，可以将fill参数设置为TRUE，在盒子中填充构件(让最大的子构件有更大的空间)：

```
gtk_box_pack_start(GTK_BOX(box),  
    child,  
    TRUE, TRUE, 0);
```

这种设置的效果见图4-4。

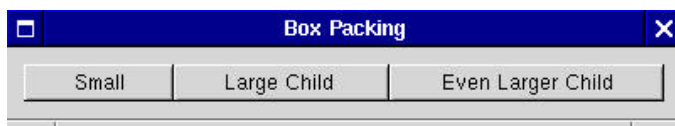


图4-4 非同质，expand = TRUE 和 fill = TRUE

3. 同质GtkBox组装模式

组装一个同质的GtkBox只有两种有意义的方法。expand参数是与同质的GtkBox是无关的，所以这两种情况是对应于fill参数的不同设置。

如果fill是FALSE，将会得到图4-5的结果。注意，组装盒逻辑划分为三个部分，但是只有最大的子构件才占据了它的整个空间。其他构件只填满了空间的三分之一。如果 fill是TRUE，

将得到图4-6所示的结果，所有的构件都是一样大的。



图4-5 同质的GtkBox，fill = FALSE

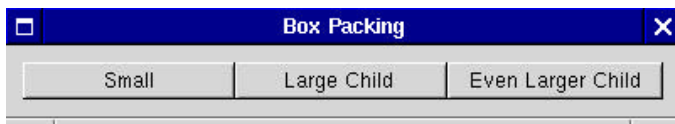


图4-6 同质的GtkBox，fill = TRUE

4. 组装摘要

图4-7同时显示了5种组装技巧。它们都是组装到一个同质的GtkVBox中，fill设置为TRUE，构件之间的间距是两个像素。从这应该能够理解它们的相对效果。要记住，还可以改变padding和spacing参数来增加或减少构件之间的空白空间。

最后一点：注意expand和fill参数只在盒子的尺寸比它要求的尺寸大时才有用。也就是说，这些参数决定额外的空间如何分配。典型情况下，当用户调整窗口的尺寸，让它比缺省尺寸大时，额外空间才显示出来。因而，应该尽量尝试调整窗口的尺寸，以保证构件是正确组装的。



图4-7 五种用组装盒组装构件的方法

4.4.4 表格构件GtkTable

GtkTable（表格构件）是很常用的用于定位的构件。我们用表格构件创建一个网格，把构件放在网格里。构件可以在网格中占据任意多个格子。

用gtk_table_new创建一个表格构件：

```
GtkWidget *gtk_table_new( gint rows,
                          gint columns,
                          gint homogeneous);
```

第一个参数是表格的行数，第二个参数是表格的列数。

homogeneous参数与表格如何划分尺寸有关。如果 homogeneous参数是TRUE，表格的所有格子都是同样大的，并且，格子的大小等于表格中最大构件的大小。如果 homogeneous是FALSE，表格的大小由同一行上最高构件和同一列最宽的构件确定。行和列从 0到n编号，其

中n是调用gtk_table_new函数创建表格时指定的行或列数。这样，如果明确指定 rows = 2和columns = 2，外观就像图4-8所示一样：

注意 坐标系统的原点在表格的左上角。

要将构件放到表格中，可以使用下列函数：

```
void gtk_table_attach( GtkTable *table,
                      GtkWidget *child,
                      gint left_attach,
                      gint right_attach,
                      gint top_attach,
                      gint bottom_attach,
                      gint xoptions,
                      gint yoptions,
                      gint xpadding,
                      gint ypadding );
```

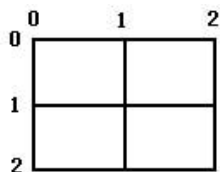


图4-8 2×2表格示意图

第一个参数“table”是前面创建的表格，第二个参数“child”是你希望放置在表格里的构件。left_attach和right_attach参数明确指定构件放在什么地方，并且占据多少个格子。

如果要将按钮放在2×2表格下面一行的右边的格子里面，并且想要它填充整个格子，可以设置为：left_attach= 1, right_attach = 2, top_attach = 1, bottom_attach = 2。

现在，如果想让一个 widget占据2×2表格中的上面一行，应该使 left_attach = 0, right_attach = 2, top_attach = 0, bottom_attach = 1。

xoptions和 yoptions用于指定组装选项，可以按“位或”设置以允许多重选项。这些选项是：

- GTK_FILL：如果表格的格子比 widget大，并且指定了GTK_FILL,构件将扩大并占满所有可用空间。
- GTK_SHRINK：如果表格比它所要求的空间还小，（通常由用户调整窗口的尺寸），那么构件会被放在窗口的底部以外的区域，无法看见。如果指定 GTK_SHRINK，构件将与表格一同缩小。
- GTK_EXPAND：让表格使用窗口的所有保留空间。

padding参数和GtkBox里的padding一样，在构件的周围产生一个空白的区域。

gtk_table_attach()函数选项较多。为了方便，有下面的快捷函数：

```
void gtk_table_attach_defaults( GtkTable *table,
                               GtkWidget *widget,
                               gint left_attach,
                               gint right_attach,
                               gint top_attach,
                               gint bottom_attach );
```

X和 Y选项默认是 GTK_FILL | GTK_EXPAND,并且X和Y的padding参数设为0，其余的参数与前面的函数一样。

我们也可以用gtk_table_set_row_spacing()和gtk_table_set_col_spacing()函数设置行、列间距。所设置的间距位于指定的行或列之间。

```
void gtk_table_set_row_spacing( GtkTable *table,
                               gint row,
```

```
        gint spacing);  
void gtk_table_set_col_spacing ( GtkTable *table,  
                                gint column,  
                                gint spacing);
```

注意 对列来说，空间位于列的右边，对行来说，它位于行的下边。

也可以为所有的行或/和列设置相同的间隔。

```
void gtk_table_set_row_spacings( GtkTable *table,  
                                gint spacing);  
void gtk_table_set_col_spacings( GtkTable *table,  
                                gint spacing);
```

注意 在上面的函数中，最后一行和最后一列并没有设置任何空间。

表格组装示例

这里我们将制作一个窗口，上面放一个 2×2 表格，在表格中放三个按钮。第一个和第二个按钮放在上面一排。第三个按钮(退出按钮)被放置在下面一排，跨越两列。

这里是源代码：

```
/* 表格示例table.c */  
  
#include <gtk/gtk.h>  
  
/* 回调函数  
 * 将传递过来的数据打印到控制台上 */  
void callback( GtkWidget *widget,  
              gpointer data )  
{  
    g_print ("Hello again - %s was pressed\n", (char *) data);  
}  
  
/* 这个回调函数退出程序 */  
void delete_event( GtkWidget *widget,  
                  GdkEvent *event,  
                  gpointer data )  
{  
    gtk_main_quit ();  
}  
  
int main( int argc,  
          char *argv[] )  
{  
    GtkWidget *window;  
    GtkWidget *button;  
    GtkWidget *table;  
  
    gtk_init (&argc, &argv);  
  
    /* 创建一个新窗口 */  
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

```
/* 设置窗口标题 */
gtk_window_set_title (GTK_WINDOW (window), "Table");

/* 设置delete_event信号的回调函数, 立即退出GTK*/
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                    GTK_SIGNAL_FUNC (delete_event), NULL);

/* 设置窗口的边框宽度 */
gtk_container_set_border_width (GTK_CONTAINER (window), 20);

/* 创建一个2×2的表格 */
table = gtk_table_new (2, 2, TRUE);

/* 将窗口放在主窗口上 */
gtk_container_add (GTK_CONTAINER (window), table);

/* 创建第一个按钮 */
button = gtk_button_new_with_label ("button 1");

/* 点击按钮时, 调用 "callback"
 * 以一个指针 "button 1"作为它的参数*/
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (callback), (gpointer) "button 1");

/* 将第一个按钮插入到表格的左上角格子中 */
gtk_table_attach_defaults (GTK_TABLE(table), button, 0, 1, 0, 1);
gtk_widget_show (button);

/* 创建第二个按钮 */
button = gtk_button_new_with_label ("button 2");
/* 点击这个按钮时, 调用 "callback"函数
 * 以指针 "button 2"为参数 */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (callback), (gpointer) "button 2");
/* 将第二个按钮插入到表格构件的右上角格子中 */
gtk_table_attach_defaults (GTK_TABLE(table), button, 1, 2, 0, 1);
gtk_widget_show (button);

/* 创建 "Quit"按钮*/
button = gtk_button_new_with_label ("Quit");

/* 点击 "Quit"按钮时, 调用 "delete_event"函数退出程序 */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (delete_event), NULL);

/* 将退出按钮插入到表格构件下面的两个格子中 */
gtk_table_attach_defaults (GTK_TABLE(table), button, 0, 2, 1, 2);

gtk_widget_show (button);
gtk_widget_show (table);
```

```

    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
/*示例结束 */

```

编译它，然后启动 table应用程序。运行结果见图 4-9。
尝试一下调整窗口的尺寸，看一看按钮如何变化。



图4-9 用表格定位构件

4.4.5 固定容器构件GtkFixed

GtkFixed(固定容器构件)允许将构件放在窗口的固定位置，这个位置是相对于窗口的左上角的。构件的位置可以动态改变。

使用GtkFixed为构件定位，在大多数情况下都是不可取的。因为当用户调整窗口尺寸时，构件不能适应窗口的尺寸变化。当然，你可以在窗口尺寸变化的时候采取行动，调整构件的位置和大小。

只有三个与固定容器构件相关的函数：

```

GtkWidget* gtk_fixed_new( void );
void gtk_fixed_put( GtkFixed *fixed,
                   GtkWidget *widget,
                   gint16 x,
                   gint16 y );
void gtk_fixed_move( GtkFixed *fixed,
                   GtkWidget *widget,
                   gint16 x,
                   gint16 y );

```

其中：

gtk_fixed_new函数用于创建新的固定容器构件。

gtk_fixed_put函数将构件放在由x, y指定的位置。

gtk_fixed_move 函数将指定构件移动到新位置。

注意，这几个函数只是将构件定位，构件将以缺省尺寸显示。如果要想指定构件的长度和宽度，可以使用下面的函数。

```

#include <gtk/gtkwidget.h>
void gtk_widget_set_uposition(GtkWidget* widget,
                             gint x,
                             gint y)
void gtk_widget_set_usize(GtkWidget* widget,
                          gint width,
                          gint height)

```

这两个参数中的参数x、y、width和height可以是-1，这时对这个参数使用缺省值。

下面的例子演示了怎样使用固定容器构件。

```

/* 固定容器示例fixed.c */
#include <gtk/gtk.h>

gint x=50;
gint y=50;

```

```
/* 这个回调函数将按钮移动到固定容器的新位置 */
void move_button( GtkWidget *widget,
                  GtkWidget *fixed )
{
    x = (x+30)%300;
    y = (y+50)%300;
    gtk_fixed_move( GTK_FIXED(fixed), widget, x, y);
}

int main( int    argc,
          char *argv[] )
{
    GtkWidget *window;
    GtkWidget *fixed;
    GtkWidget *button;
    gint i;

    /* 初始化GTK */
    gtk_init(&argc, &argv);

    /* 创建一个新按钮 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Fixed Container");

    /* 为窗口的"destroy"事件设置一个信号处理函数 */
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (gtk_main_quit), NULL);

    /* 设置窗口的边框宽度 */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个固定容器构件 */
    fixed = gtk_fixed_new();
    gtk_container_add(GTK_CONTAINER(window), fixed);

    gtk_widget_show(fixed);

    for (i = 1 ; i <= 3 ; i++) {
        /* 创建一个标题为"Press Me"的新按钮 */
        button = gtk_button_new_with_label ("Press me");

        /* 当按钮接收到"clicked"信号时, 调用move_button()函数 */
        gtk_signal_connect (GTK_OBJECT (button), "clicked",
                            GTK_SIGNAL_FUNC (move_button), fixed);

        /* 将按钮组装到一个固定容器构件中 */
        gtk_fixed_put (GTK_FIXED (fixed), button, i*50, i*50);

        /* 最后一步是显示新建的构件 */
        gtk_widget_show (button);
    }
}
```



```
}

/* 显示窗口 */
gtk_widget_show (window);

/* 进入主循环 */
gtk_main ();

return(0);
}

/* 示例结束 */
```

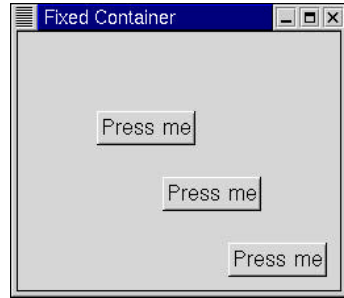


图4-10 GtkFixed构件

上面的代码的执行效果见图4-10。点击“Press Me”按钮，按钮会在窗口上移动（实际上是在GtkFixed上移动）。

4.4.6 布局容器构件GtkLayout

GtkLayout(布局容器构件)与固定容器构件类似，不过它可以在一个无限的滚动区域定位构件(其实也不能大于4294967296像素)。在X系统中，窗口的宽度和高度只能限于在32767像素以内。布局容器构件使用一些特殊的技巧越过这种限制。所以，当在滚动区域内有很多子构件时，可以让滚动更平滑。

用以下函数创建布局容器构件：

```
GtkWidget *gtk_layout_new( GtkAdjustment *hadjustment,
                           GtkAdjustment *vadjustment );
```

可以看到指定布局容器构件滚动时要使用的调整对象。关于“调整对象”，我们在后面的章节另有介绍。

用以下函数在布局容器构件内添加和移动子构件。将构件 widget 放在布局容器构件中坐标为 x、y 的位置：

```
void gtk_layout_put( GtkLayout *layout,
                    GtkWidget *widget,
                    gint      x,
                    gint      y );
```

将构件 widget 移动到布局容器构件中坐标为 x、y 的位置：

```
void gtk_layout_move( GtkLayout *layout,
                     GtkWidget *widget,
                     gint      x,
                     gint      y );
```

布局容器构件的尺寸可以用以下函数指定：

```
void gtk_layout_set_size( GtkLayout *layout,
                         guint      width,
                         guint      height );
```

布局容器构件是 Gtk 构件中一种会重绘自身的构件，当使用上面的函数改变了构件内的对象时，它会在屏幕上重绘自身。当想对布局容器构件做较大的变化时，可以用下面的函数“冻结”或“解冻”该构件，这样会禁止或启用布局容器构件重绘自身，能防止窗口闪烁。

“冻结”布局容器构件：

第5章 按钮构件

5.1 普通按钮GtkButton

GtkButton(普通按钮构件)是应用程序中使用最广泛的构件。它一般用于当用户点击它时执行某个动作。因而，如果某个构件能够对点击进行响应，实际上可以将它用作按钮。按钮构件的创建和使用相当简单。

前面已经介绍过，GtkButton是从GtkBin派生而来的构件，它本身就是一个容器。因而，可以充分利用这一点来创建非常丰富的按钮类型。比如在按钮上放置一幅图片、动画等。甚至可以创建非常古怪的按钮。

按钮构件也是一个基类，GtkToggleButton（开关按钮）、GtkCheckButton（检查按钮）等构件都是从GtkButton派生而来的。

有两种创建按钮的方法。可以用 `gtk_button_new_with_label()` 创建带标题的按钮，或用 `gtk_button_new()` 创建空白按钮，然后可以将标签和 pixmap 图片组装到新按钮中。要这么做，需要先创建一个新的 GtkBox，并用常用的 `gtk_box_pack_start()` 函数将对象（文本或图片）组装到这个 GtkBox 里，然后用 `gtk_container_add` 函数将 box 组装到按钮里。

下面是用 `gtk_button_new` 创建一个带图片和标题的按钮的例子。此处将创建一个 GtkBox，且放置图片和标题的部分和其余部分分开了。可以将这段代码用在程序的其他地方。关于 pixmaps 的用法在本教程的后面有更进一步的例子。

```
/* 按钮示例开始 buttons.c */

#include <gtk/gtk.h>

/* 创建一个新的 GtkHBox 组装盒，其中带一个标签和一幅图片
 * 然后返回这个组装盒 */

GtkWidget *xpm_label_box( GtkWidget *parent,
                           gchar      *xpm_filename,
                           gchar      *label_text )
{
    GtkWidget *box1;
    GtkWidget *label;
    GtkWidget *pixmapwid;
    GdkPixmap *pixmap;
    GdkBitmap *mask;
    GtkStyle *style;

    /* 为 xpm 图片和标签创建组装盒 */
    box1 = gtk_hbox_new (FALSE, 0);
    gtk_container_set_border_width (GTK_CONTAINER (box1), 2);
```

```
/* 从父构件中取得风格参数以设置组装盒的背景颜色等风格 */
style = gtk_widget_get_style(parent);

/* 下载获得要填充的xpm图片*/
pixmap = gdk_pixmap_create_from_xpm (parent->window, &mask,
                                     &style->bg[GTK_STATE_NORMAL],
                                     xpm_filename);
pixmapwid = gtk_pixmap_new (pixmap, mask);

/*为按钮创建一个标签 */
label = gtk_label_new (label_text);

/* 将标签和图片组装到GtkHBox中*/
gtk_box_pack_start (GTK_BOX (box1),
                    pixmapwid, FALSE, FALSE, 3);

gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 3);
gtk_widget_show(pixmapwid);
gtk_widget_show(label);

return(box1);
}

/* 常见的回调函数 */
void callback( GtkWidget *widget,
              gpointer data )
{
    g_print ("Hello again - %s was pressed\n", (char *) data);
}

int main( int argc,
          char *argv[] )
{
    /* 所有构件的存储类型都是GtkWidget */
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box1;

    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    /* 设置窗口的标题 */
    gtk_window_set_title (GTK_WINDOW (window), "Pixmap'd Buttons!");

    /* 最好对所有的窗口都做下面的工作 */
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                       GTK_SIGNAL_FUNC (gtk_exit), NULL);
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                       GTK_SIGNAL_FUNC (gtk_exit), NULL);
    /* 设置窗口的边框宽度 */
}
```

```

gtk_container_set_border_width (GTK_CONTAINER (window), 10);
gtk_widget_realize(window);

/* 创建一个新按钮 */
button = gtk_button_new ();

/* 将按钮的"clicked"信号连接到前面创建的回调函数上 */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (callback),
                    (gpointer) "cool button");

/* 调用我们定义的组装箱创建函数以创建一个带图片和标签的盒子 */
box1 = xpm_label_box(window, "info.xpm", "cool button");

/* 组装所有的构件。并显示它们 */
gtk_widget_show(box1);
gtk_container_add (GTK_CONTAINER (button), box1);
gtk_widget_show(button);
gtk_container_add (GTK_CONTAINER (window), button);
gtk_widget_show (window);
/* 开始程序的主循环，等待用户的动作 */
gtk_main ();
return(0);
}
/*示例结束 */

```

将上面的源代码保存为 button.c 文件，然后写一个 Makefile 文件，如下所示：

```

CC = gcc
buttons: buttons.c
    $(CC) `gtk-config --cflags` buttons.c -o buttons `gtk-config --libs`
clean:
    rm -f *.o buttons

```

在 shell 提示符下输入 make 命令，编译该示例，然后输入 ./button 就可执行这个程序。当点击图 5-1 所示的 cool button 按钮时，会在终端上显示 “Hello again Coll Button was pressed” 信息。

上面的示例中使用了一个函数 xpm_label_box()，它将一幅图片和一个标签构件放到一个 GtkWidget 里面，这样可以创建一个带图片和标签的按钮。这个函数也可以用在其他容器构件上。

注意，要了解在 xpm_label_box 函数中是怎么调用 gtk_widget_get_style 的。每个构件都有自己的“风格”，

包括各种情形下的前景色和背景颜色、字体以及其他与构件有关的图像数据。这些风格在每个构件中都有默认值，并且许多 GDK 函数调用都需要这些值。比如 gdk_pixmap_create_from_xpm 函数，在这里给定了“正常”的背景色。

还要注意到：设置窗口的边框宽度后，调用了 gtk_widget_realize 函数。这个函数用 GDK 创建与构件相关的 X 窗口。当一个构件调用 gtk_widget_show() 函数显示该构件时，会自动调用这个函数。因而前面的例子里面没有使用这个函数。但是调用 gdk_pixmap_create_from_xpm 时需要它的窗口参数指向一个实际存在的 X 窗口，因而在使用这个 GDK 调用前必须先调用

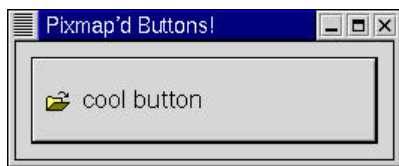


图5-1 按钮构件

gtk_widget_realize函数显现该构件。

按钮构件有下列信号：

pressed 当鼠标按键在按钮构件内按下时引发。

released 当鼠标按键在按钮构件内部松开时引发。

clicked 当鼠标按键在按钮构件上按下并松开时引发。

enter 当鼠标按键进入按钮构件时引发。

leave 当鼠标按键离开按钮构件时引发。

有时候，用户没有点击按钮，但是需要执行点击按钮时所对应的动作。一种方法是直接调用按钮的clicked信号的回调函数，另一种方法就是调用一个函数，让这个函数引发 clicked信号，这样自然就会调用这个回调函数。对鼠标按下、松开、进入和离开等信号也有类似的函数。

对给定按钮构件button引发pressed信号，效果是直接调用pressed信号对应的回调函数：

```
void gtk_button_pressed (GtkButton *button);
```

对给定按钮构件button引发released信号：

```
void gtk_button_released (GtkButton *button);
```

对给定按钮构件button引发clicked信号：

```
void gtk_button_clicked (GtkButton *button);
```

对给定按钮构件button引发enter信号：

```
void gtk_button_enter (GtkButton *button);
```

对给定按钮构件button引发leave信号：

```
void gtk_button_leave (GtkButton *button);
```

如果想让按钮构件不能接受点击，也就是不对用户响应，可以将构件设置为“不敏感”的。这在许多情况下都是很有作用的。比如，应用程序必须按照某个流程执行，否则就会出错。这时不应该寄希望于用户会按照正确的步骤执行，而是应该让用户根本就不能点击它。先将按钮设置为“不敏感”，然后当满足特定条件时，再将按钮设置为“敏感”的。

下面的函数设置构件的敏感性。将sensitive设置为TRUE，按钮可以接受用户点击，否则，按钮是灰色的，不能对用户的点击响应。

```
void gtk_widget_set_sensitive (GtkWidget *widget,  
                               gboolean sensitive);
```

使用这个函数可以设置其他构件的敏感性。

5.2 开关按钮 GtkToggleButton

GtkToggleButton(开关按钮构件)是从普通按钮里派生出来的。它们的状态总是处于两种状态中的一种：按下或弹起。其他方面和普通按钮非常相似。它的状态可以是被按下，当你再次点击时，将向上弹起。再点击一次，它又会被按下去。

开关按钮是检查按钮(check button)和无线按钮(radio button)的基础。与此类似，无线和检查按钮继承了开关按钮的许多函数调用。

创建一个新GtkToggleButton：

```
GtkWidget *gtk_toggle_button_new(void);
```



```
GtkWidget *gtk_toggle_button_new_with_label( gchar *label );
```

这两个函数与GtkButton的函数工作原理是一样的。第一个函数创建空的开关按钮，第二个函数创建一个带标签的开关按钮。

可以使用下例所示的代码取得开关按钮（包括无线和检查按钮）的状态。这个函数先使用GTK_TOGGLE_BUTTON宏将构件转换为指向开关按钮的指针，然后访问开关按钮的结构的active域的值，以检测开关按钮的状态。对开关按钮（包含它的派生构件检查按钮和无线按钮），我们最感兴趣的信号是 toggled。要检查这些按钮的状态，设置一个信号处理函数以捕获 toggled信号，然后访问按钮结构中的 active域以判定它的状态值。“toggled”信号的回调函数应该是下面这个样子：

```
void toggle_button_callback (GtkWidget *widget, gpointer data)
{
    if (GTK_TOGGLE_BUTTON (widget)->active)
    {
        /* 如果执行这段程序，表明按钮是按下的 */
    } else {
        /* 如果执行这段程序，按钮是弹起的 */
    }
}
```

要强行设置一个开关按钮（以及它的派生构件：无线按钮和检查按钮）的状态，使用下面的函数：

```
void gtk_toggle_button_set_active( GtkToggleButton *toggle_button,
                                   gint                state );
```

上面的函数适用于开关按钮以及检查按钮和无线按钮。将要设置的按钮作为第一个参数，第二个参数设置为TRUE或FALSE，明确指定它是按下（压下或选中）或弹起（未选中）的。缺省是向上或FALSE。

注意，如果调用gtk_toggle_button_set_active()函数，并且按钮的状态发生了实际变化，按钮会引发一个clicked信号。

```
void gtk_toggle_button_toggled (GtkToggleButton *toggle_button);
```

这个函数简单切换按钮的状态，并引发一个 toggled信号。

5.3 检查按钮 GtkCheckButton

GtkCheckButton(检查按钮构件)从上面介绍的开关按钮继承了许多属性和函数，但是外观上略有不同。检查按钮的文本标签是在按钮的旁边，而开关按钮的则在按钮里面。检查按钮经常用于在应用程序中切换“开启”或者“关闭”选项。

创建GtkCheckButton的函数如下：

```
GtkWidget *gtk_check_button_new(void );
GtkWidget *gtk_check_button_new_with_label ( gchar * label );
```

第二个以new_with_label结尾的函数创建一个带标签的检查按钮。

获取检查按钮的状态的方法与前述的开关按钮完全一样。

5.4 无线按钮 GtkRadioButton

GtkRadioButton(无线按钮构件)与检查按钮类似，不同之处在于无线按钮是分组的，同一

组内的按钮一次只能有一个被选中，也就是说它们是互斥的。当需要从一个较小的选项列表中选择一项时，使用无线按钮非常合适。

用下面的函数创建新的 GtkRadioButton：

```
GtkWidget *gtk_radio_button_new( GSList *group );

GtkWidget *gtk_radio_button_new_with_label( GSList *group,
                                             gchar *label );
```

可以看到，上面的两个函数都有一个 group 参数。因为无线按钮必须分组，所以必须指定按钮属于哪一个组以便它们能够正常工作。

第一次调用 gtk_radio_button_new_with_label 或 gtk_radio_button_new_with_label 函数时应给 group 参数传递 NULL 值，然后用下面的函数创建一个组：

```
GSList *gtk_radio_button_group( GtkWidget *radio_button );
```

要注意，必须对每个新建的无线按钮调用 gtk_radio_button_group 函数，并将其加到按钮组中，其中的参数就是要加入的按钮。然后将其结果传递到下一个调用 gtk_radio_button_new 或 gtk_radio_button_new_with_label 的函数中。这样可以创建一系列的按钮。下面的示例详细介绍了这些内容。可以用下面的语法缩短这个过程。这种语法不需要维护一个按钮列表。在示例中用这种形式创建了第三个无线按钮：

```
button2 = gtk_radio_button_new_with_label(
    gtk_radio_button_group (GTK_RADIO_BUTTON (button1)),
    "button2");
```

最好在创建按钮时明确指出哪一个按钮是缺省选中的按钮。用下面的方法：

```
void gtk_toggle_button_set_active( GtkWidget *toggle_button,
                                   gint state );
```

这个函数在 GtkToggleButton 中已经介绍过了，在这里按同样的方法使用。一旦无线按钮已经分组，一组按钮中就只能有一个处于活动状态（被选中）。如果用户点击某个无线按钮，然后点击同组的另一个无线按钮，前一个无线按钮会首先引发一个 toggled 信号（报告它变成不活动的），然后第二个按钮会引发一个 toggled 信号（报告它变成活动按钮）。

使用上面的方法，我们能够在窗口上添加几个无线按钮，将它们分为几组，让同组之间是互斥的，不同组之间互不相干。最好能用 GtkFrame 或者类似的构件将它们从布局上分开。否则，用户可能会误操作。

下列例子将创建一个含有三个无线按钮的按钮组。

```
/* 无线示例开始 radiobuttons.c */
#include <gtk/gtk.h>
#include <glib.h>

void close_application( GtkWidget *widget,
                        GdkEvent *event,
                        gpointer data )

{
    gtk_main_quit();
}

int main( int argc, char *argv[] )
{
    GtkWidget *window = NULL;
    GtkWidget *box1;
```

```
GtkWidget *box2;
GtkWidget *button;
GtkWidget *separator;
GSList *group;

gtk_init(&argc,&argv);
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                    GTK_SIGNAL_FUNC(close_application),
                    NULL);

gtk_window_set_title (GTK_WINDOW (window), "radio buttons");
gtk_container_set_border_width (GTK_CONTAINER (window), 0);

box1 = gtk_vbox_new (FALSE, 0);
gtk_container_add (GTK_CONTAINER (window), box1);
gtk_widget_show (box1);
box2 = gtk_vbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

button = gtk_radio_button_new_with_label (NULL, "button1");
gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
gtk_widget_show (button);
group = gtk_radio_button_group (GTK_RADIO_BUTTON (button));
button = gtk_radio_button_new_with_label(group, "button2");
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);
gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
gtk_widget_show (button);
button = gtk_radio_button_new_with_label
    (gtk_radio_button_group (GTK_RADIO_BUTTON (button)),
     "button3");

gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
gtk_widget_show (button);
separator = gtk_hseparator_new ();
gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 0);
gtk_widget_show (separator);
box2 = gtk_vbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, TRUE, 0);
gtk_widget_show (box2);
button = gtk_button_new_with_label ("close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC(close_application),
                           GTK_OBJECT (window));
gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
gtk_widget_grab_default (button);
```

```
gtk_widget_show (button);  
gtk_widget_show (window);  
gtk_main();  
return(0);  
}  
/* 示例结束 */
```

将上面的代码保存为 `radiobuttons.c`，然后写一个像下面这样的 Makefile 文件。在 shell 提示符下输入 `make` 命令进行编译。

```
CC = gcc  
radiobuttons: radiobuttons.c  
    $(CC) `gtk-config --cflags` radiobuttons.c -oradiobuttons `\  
        gtk-config --libs`  
clean:  
    rm -f *.o radiobuttons
```

运行结果如图 5-2 所示。在 `button1`、`button2` 和 `button3` 上点击，看一下它们的状态有什么变化。这几个按钮应该是互斥的。

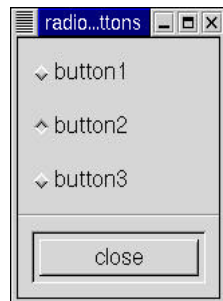


图5-2 无线按钮示例

第6章 调整对象

GTK有多种构件能够由用户通过鼠标或键盘进行调整，比如范围构件。还有一些构件，比如说GtkText和GtkViewport，内部都有一些可调整的属性。

很明显，当用户调整范围构件的值时，应用程序需要对值的变化进行响应。一种办法就是当构件的调整值发生变化时，让每个构件引发自己的信号，将新值传递到信号处理函数中，或者让它在构件的内部数据结构中查找构件的值。但是，也许需要将这个调整值同时连接到几个构件上，使得调整一个值时，其他的值也随之变化。最明显的例子就是将一个滚动条连接到一个视角构件或者滚动的文本区上。如果每个构件都有自己的设置或获取调整值的方法，程序员或许需要自己编写很复杂的信号处理函数，以便将这些不同构件之间的变化同步或相关联。

GTK用一个调整对象解决了这个问题。调整对象不是构件，但是为构件提供了一种以抽象、灵活的方法传递调整值信息的方法。调整对象最明显的用处就是为范围构件（比如滚动条和比例构件）储存配置参数和值。然而，因为调整对象是从 GtkObject派生的，在其正常的数据结构之外，它还具有一些特殊的功能。最重要的是，它们能够引发信号，就像构件一样，这些信号不仅能够让程序对用户可在调整构件上的输入进行响应，还能在可调整构件之间透明地传播调整值。

在许多其他的构件中都能够看到调整对象的用处。比如进度条、视角、滚动窗口等。

6.1 创建一个调整对象

许多使用调整对象的构件都能够自动创建它，但是有些情况下，必须自己手工创建。用下面的函数创建调整对象：

```
GtkObject *gtk_adjustment_new( gfloat value,
                                gfloat lower,
                                gfloat upper,
                                gfloat step_increment,
                                gfloat page_increment,
                                gfloat page_size );
```

其中的“value”参数是要赋给调整对象的初始值，通常对应于一个可调整构件的最高或最低位置。“lower”参数指定调整对象能取的最低值，“step_increment”参数指定用户能小步增加的值，“page_increment”是用户能大部调整的值。“page_size”参数通常用于分栏构件的可视区域。“upper”参数用于表示分栏构件的子构件的最底部或最右边的坐标。因而，它不一定总是“value”能取的最大值，因为这些构件的“page_size”通常是非零值。

6.2 使用调整对象

可调整构件大致可以分为两组：一组对这些值使用特定的单位，另一组将这些值当作任意数值。后一组包括范围构件：滚动条、比例构件、进度条以及微调按钮。这些构件的值都

可以使用鼠标和键盘直接进行调整。它们将调整对象的 upper和lower值当作用户能够操纵的调整值的范围。缺省时，它们只会修改调整对象的值（也就是说，它们的范围一般都是不变的）。

另一组包含文本构件、视角构件以及滚动窗口构件。所有这些构件都是间接通过滚动条进行调整的。所有使用调整对象的构件都可以使用自己的调整对象，或者使用外部创建的调整对象，但是最好让这一类构件都使用它们自己的调整对象。

现在，你也许在想，文本构件和视角构件除了调整对象的值以外，其他的值都是由它控制的，而滚动条就是控制调整值的，如果在滚动条和文本构件之间共享调整对象，操纵滚动条会自动调整文本构件吗？确实如此，就像下面的代码所做的：

```
/* 创建自己的调整对象 */
text = gtk_text_new (NULL, NULL);

/*让垂直滚动条使用文本构件自己创建的调整对象 */
vscrollbar = gtk_vscrollbar_new (GTK_TEXT(text)->vadj);
```

6.3 调整对象内部机制

如果我想创建一个信号处理函数，当用户调整范围构件或微调按钮时让这个处理函数进行响应，应该从调整对象中取什么值，怎样从中取值呢？要解决这个问题，先看一眼

_GtkAdjustment结构的定义：

```
struct _GtkAdjustment
{
    GtkData data;
    gfloat lower;
    gfloat upper;
    gfloat value;
    gfloat step_increment;
    gfloat page_increment;
    gfloat page_size;
};
```

应该知道的第一件事就是没有宏或存取函数能够从一个调整对象中取值。所以，必须自己完成这件工作。

因为设置调整对象的值时，通常想改变它的值以适应任何使用这个调整对象的构件，GTK提供了下面的函数：

```
void gtk_adjustment_set_value( GtkAdjustment *adjustment,
                               gfloat          value );
```

前面说过，和其他构件一样，调整对象是 GObject的子类，因而，它也能够引发信号。这也是为什么当滚动条和其他可调整构件共享调整对象时它们能够自动更新的原因。所有的可调整构件都为它们的调整对象的 value_changed信号设置了一个信号处理函数。下面是这个信号在_GtkAdjustmentClass结构中的定义：

```
void (* value_changed) (GtkAdjustment *adjustment);
```

各种使用调整对象的构件都会当它们的值发生变化时引发它们的调整对象的信号。这种情况发生在当用户输入值使范围构件的滑块移动和当程序使用 gtk_adjustment_set_value()函数显式地改变调整对象的值时。所以，如果有一个比例构件，想在它的值改变时改变一幅画的

旋转角度，应该创建像下面这样的回调函数：

```
void cb_rotate_picture (GtkAdjustment *adj, GtkWidget *picture)
{
    set_picture_rotation (picture, adj->value);
    ...
}
```

将这个回调函数连接到构件的调整对象上：

```
gtk_signal_connect (GTK_OBJECT (adj), "value_changed",
                   GTK_SIGNAL_FUNC (cb_rotate_picture), picture);
```

当构件重新配置了它的调整对象的 upper和lower值时（比如，用户向文本构件添加了更多的文本时），发生了什么？在这种情况下，它会引发一个“changed”信号：

```
void (* changed) (GtkAdjustment *adjustment);
```

范围构件一般为这个信号设置回调函数，构件会改变它们的外观以反映变化。例如，滚动条上的滑块大小会根据它的调整对象的变化而变化。

一般不使用这个信号，除非想要写一个新的范围构件。不过，如果直接改变了调整对象的值，应该引发这个信号，以便重新配置它。用下面的函数引发这个信号：

```
gtk_signal_emit_by_name (GTK_OBJECT (adjustment), "changed");
```


第7章 文本构件 GtkText

GtkText (文本构件) 允许多行显示或编辑文本。它支持多种颜色以及多种字体的文本, 允许它们以任何需要的形式混合显示, 还有许多与 Emacs兼容的文本编辑命令。

文本构件支持完全的剪切/粘贴功能, 还包括双击选择一个单词和三击选择整行的功能。

注意, 请将GtkText和GtkEntry构件区分开。GtkEntry只能显示或编辑一行字符串, 而不能将多种字体和多种颜色的文本混排。

7.1 创建、配置文本构件

创建新Text构件只有一个函数:

```
GtkWidget *gtk_text_new( GtkAdjustment *hadj,  
                          GtkAdjustment *vadj );
```

其中的参数允许为文本构件指定水平和垂直的调整对象, 并且可以用于跟踪构件的位置。向gtk_text_new函数传递NULL, 函数会为文本构件创建自己的调整对象。

```
void gtk_text_set_adjustments( GtkText *text,  
                              GtkAdjustment *hadj,  
                              GtkAdjustment *vadj );
```

上面的函数可以随时改变文本构件的水平和垂直的调整对象。

当文本构件中的文本超过构件能显示的空间时, 文本构件不会自动显示滚动条。所以我们必须另行创建滚动条, 将它们添加到要显示的窗口布局上。

```
vscrollbar = gtk_vscrollbar_new (GTK_TEXT(text)->vadj);  
gtk_box_pack_start(GTK_BOX(hbox), vscrollbar, FALSE, FALSE, 0);  
gtk_widget_show (vscrollbar);
```

上面的小段代码创建了一个垂直滚动条, 并将它添加到文本构件的垂直 adjustment构件上, 然后将它们组装到一个“ 组装箱 ”中。但是文本构件目前不支持水平滚动条。

文本构件有两个主要用途: 允许用户编辑一段文本, 或向用户显示多行文本。为了在两种操作模式之间进行切换, 文本构件有以下函数:

```
void gtk_text_set_editable( GtkText *text,  
                            gint     editable );
```

其中, editable参数可以是TRUE或FALSE, 它指定用户是否可以编辑文本内容。当 Text构件是可编辑的时, 会在当前插入点显示一个光标。

当然, 不仅可以使使用文本构件的这两种模式。还可以随时切换构件的可编辑模式, 随时插入文本。

文本构件在文本如果太长, 一行显示不下时会换行。缺省方式是在单词之间分行, 可以用以下函数将其改变:

```
void gtk_text_set_word_wrap( GtkText *text,  
                             gint     word_wrap );
```

这个函数允许我们指定文本构件是否在单词之间换行。 word_wrap参数的值可以是TRUE

或FALSE。

7.2 操作文本

可以用以下函数设置文本构件的插入点：

```
void gtk_text_set_point( GtkText *text,
                        guint      index );
```

index参数是要设置插入点的位置。

与上面的函数类似，使用下面的函数可以获得当前的插入点：

```
guint gtk_text_get_point( GtkText *text );
```

下面的函数可以与上面的函数联合应用：

```
guint gtk_text_get_length( GtkText *text );
```

返回当前文本的长度。长度是整个文本的字符数，其中还包括换行符等。

为了在当前插入点插入文本，可以使用 `gtk_text_insert`函数。插入时可以指定文本的背景色、前景色和字体。

```
void gtk_text_insert( GtkText      *text,
                     GdkFont      *font,
                     GdkColor      *fore,
                     GdkColor      *back,
                     const char    *chars,
                     gint           length );
```

向fore、back、font中传递NULL参数让插入的文本使用构件内部的颜色和字体设置。设置length参数为-1，将字符串全部插入。

文本构件是一种动态重绘自身的构件，它会在`gtk_main()`函数之外重绘构件。这意味着文本构件内的所有变化都会立即生效。如果文本构件内的变化很多时，可能会引起闪烁。要在文本构件内的文本变化较大时不让构件重绘，可以先“冻结”构件，临时停止动态重绘本身。构件内的更新结束时，再将构件“解冻”。

下面两个函数“冻结”、“解冻”文本构件：

```
void gtk_text_freeze( GtkText *text );
void gtk_text_thaw(  GtkText *text );
```

用以下函数删除当前位置以前或以后的 `nchars`个字符。返回值TRUE或FALSE指明操作成功或失败。

```
gint gtk_text_backward_delete( GtkText *text,
                              guint     nchars );
gint gtk_text_forward_delete ( GtkText *text,
                              guint     nchars );
```

如果要从文本构件取得文本的内容，`GTK_TEXT_INDEX(t, index)`宏可以取得 `t`构件指定位置index处的字符。

用下面的函数取得大段文本：

```
gchar *gtk_editable_get_chars( GtkEditable *editable,
                              gint          start_pos,
                              gint          end_pos );
```

这实际上是文本构件的父类的函数。end_pos设为-1指明文本的尾部。注意，索引值是从0

开始的。

这个函数为文本分配一段内存，所以使用结束后别忘了用 `g_free` 释放内存。

7.3 键盘快捷键

文本构件有许多预设的键盘快捷键，可用于常用的编辑、移动和选择等功能。它们常用 `Ctrl` 和 `Alt` 与其他键的组合键。

另外，按住 `Ctrl` 键的同时按方向键会让光标以单词为单位移动，而不是一个个地移动字符。按住 `Shift` 然后按方向键会选中或取消选择。

1. 移动快捷键

`Ctrl-A` 移到行头

`Ctrl-E` 移到行尾

`Ctrl-N` 移到下一行

`Ctrl-P` 移到前一行

`Ctrl-B` 向后一个字符

`Ctrl-F` 向前一个字符

`Alt-B` 向后一个单词

`Alt-F` 向前一个单词

2. 编辑快捷键

`Ctrl-H` 删除前一个字符(退格键)

`Ctrl-D` 删除下一个字符(删除键)

`Ctrl-W` 删除后一个单词

`Alt-D` 删除下一个单词

`Ctrl-K` 删除到行尾

`Ctrl-U` 删除一行

3. 选择快捷键

`Ctrl-X` 剪切到剪贴板

`Ctrl-C` 复制到剪贴板

`Ctrl-V` 从剪贴板粘贴

7.4 GtkText示例

```
/* 文本构件示例 text.c */
/* text.c */
#include <stdio.h>
#include <gtk/gtk.h>

void text_toggle_editable (GtkWidget *checkboxbutton,
                           GtkWidget *text)
{
    gtk_text_set_editable(GTK_TEXT(text),
                           GTK_TOGGLE_BUTTON(checkboxbutton)->active);
}
```

```
void text_toggle_word_wrap (GtkWidget *checkboxbutton,
                             GtkWidget *text)
{
    gtk_text_set_word_wrap(GTK_TEXT(text),
                           GTK_TOGGLE_BUTTON(checkboxbutton)->active);
}

void close_application( GtkWidget *widget, gpointer data )
{
    gtk_main_quit();
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *box1;
    GtkWidget *box2;
    GtkWidget *hbox;
    GtkWidget *button;
    GtkWidget *check;
    GtkWidget *separator;
    GtkWidget *table;
    GtkWidget *vscrollbar;
    GtkWidget *text;
    GdkColormap *cmap;
    GdkColor color;
    GdkFont *fixed_font;

    FILE *infile;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_usize (window, 600, 500);
    gtk_window_set_policy (GTK_WINDOW(window), TRUE, TRUE, FALSE);
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                       GTK_SIGNAL_FUNC(close_application),
                       NULL);
    gtk_window_set_title (GTK_WINDOW (window), "Text Widget Example");
    gtk_container_set_border_width (GTK_CONTAINER (window), 0);
    box1 = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), box1);
    gtk_widget_show (box1);

    box2 = gtk_vbox_new (FALSE, 10);
    gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
    gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
    gtk_widget_show (box2);

    table = gtk_table_new (2, 2, FALSE);
    gtk_table_set_row_spacing (GTK_TABLE (table), 0, 2);
```

```
gtk_table_set_col_spacing (GTK_TABLE (table), 0, 2);
gtk_box_pack_start (GTK_BOX (box2), table, TRUE, TRUE, 0);
gtk_widget_show (table);

/*创建GtkText构件*/
text = gtk_text_new (NULL, NULL);
gtk_text_set_editable (GTK_TEXT (text), TRUE);
gtk_table_attach (GTK_TABLE (table), text, 0, 1, 0, 1,
                  GTK_EXPAND | GTK_SHRINK | GTK_FILL,
                  GTK_EXPAND | GTK_SHRINK | GTK_FILL, 0, 0);
gtk_widget_show (text);

/*给GtkText构件添加垂直滚动条*/
vscrollbar = gtk_vscrollbar_new (GTK_TEXT (text)->vadj);
gtk_table_attach (GTK_TABLE (table), vscrollbar, 1, 2, 0, 1,
                  GTK_FILL, GTK_EXPAND | GTK_SHRINK | GTK_FILL, 0, 0);
gtk_widget_show (vscrollbar);

/* 取得系统颜色映射, 将映射设置为红色 */
cmap = gdk_colormap_get_system();
color.red = 0xffff;
color.green = 0;
color.blue = 0;
if (!gdk_color_alloc(cmap, &color)) {
    g_error("couldn't allocate color");
}

/* 加载固定字体 */
fixed_font = gdk_font_load ("-misc-fixed-medium-r-*-*-*140-*-*-*-*-*");

/* 实现文本构件
 * 可以插入一些文本了 */
gtk_widget_realize (text);

/*冻结text构件, 准备多行更新*/
gtk_text_freeze (GTK_TEXT (text));

/* Insert some colored text */
gtk_text_insert (GTK_TEXT (text), NULL, &text->style->black, NULL,
                 "Supports ", -1);
gtk_text_insert (GTK_TEXT (text), NULL, &color, NULL,
                 "colored ", -1);
gtk_text_insert (GTK_TEXT (text), NULL, &text->style->black, NULL,
                 "text and different ", -1);
gtk_text_insert (GTK_TEXT (text), fixed_font, &text->style->black, NULL,
                 "fonts\n\n", -1);

/* 将text.c文件加载到text窗口*/

infile = fopen("text.c", "r");
if (infile) {
```



```
char buffer[1024];
int nchars;

while (1)
{
    nchars = fread(buffer, 1, 1024, infile);
    gtk_text_insert (GTK_TEXT (text), fixed_font, NULL,
                    NULL, buffer, nchars);

    if (nchars < 1024)
        break;
}

fclose (infile);
}

/* 将text构件"解冻", 让变化显示出来*/
gtk_text_thaw (GTK_TEXT (text));

hbox = gtk_hbutton_box_new ();
gtk_box_pack_start (GTK_BOX (box2), hbox, FALSE, FALSE, 0);
gtk_widget_show (hbox);

check = gtk_check_button_new_with_label("Editable");
gtk_box_pack_start (GTK_BOX (hbox), check, FALSE, FALSE, 0);
gtk_signal_connect (GTK_OBJECT(check), "toggled",
                    GTK_SIGNAL_FUNC(text_toggle_editable), text);
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(check), TRUE);
gtk_widget_show (check);
check = gtk_check_button_new_with_label("Wrap Words");
gtk_box_pack_start (GTK_BOX (hbox), check, FALSE, TRUE, 0);
gtk_signal_connect (GTK_OBJECT(check), "toggled",
                    GTK_SIGNAL_FUNC(text_toggle_word_wrap), text);
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(check), FALSE);
gtk_widget_show (check);

separator = gtk_hseparator_new ();
gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 0);
gtk_widget_show (separator);

box2 = gtk_vbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, TRUE, 0);
gtk_widget_show (box2);

button = gtk_button_new_with_label ("close");
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC(close_application),
                    NULL);
gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
```

```
GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
gtk_widget_grab_default (button);
gtk_widget_show (button);

gtk_widget_show (window);

gtk_main ();
return(0);
}
/*示例结束*/
```

将上述代码保存为text.c，然后写一段向下面这样的Makefile文件：

```
CC = gcc
text: text.c
    $(CC) `gtk-config --cflags` text.c -o text `gtk-config --libs`
clean:
    rm -f *.o text
```

编译后，执行结果如图7-1所示。Editable检查按钮在按下时，文本构件内是可编辑的，如果该按钮是弹起的，文本构件内的文本是只读的。选中 Word Wrap检查按钮，文本会自动换行。

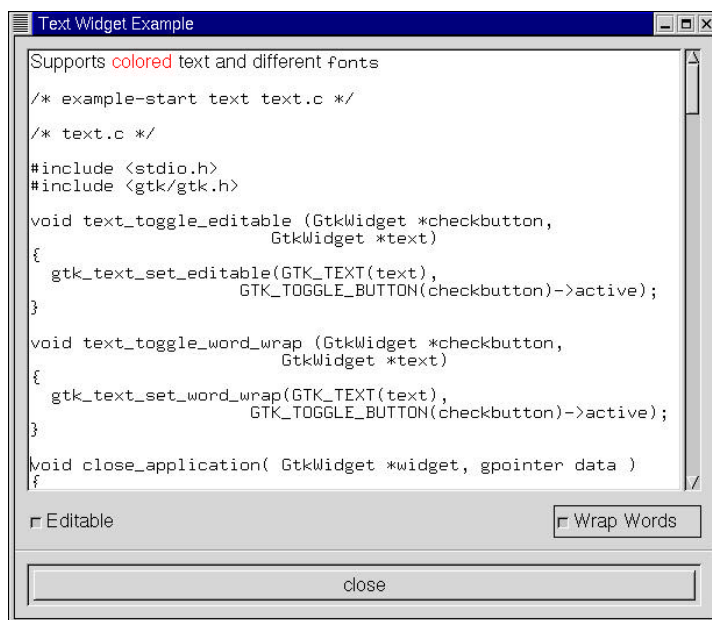


图7-1 文本构件示例

第8章 范围构件 GtkRange

GtkRange(范围构件)是一大类构件，包含常见的滚动条构件和较少见的“比例”构件。尽管这两种构件是用于不同的目的，它们在功能和实现上都是非常相似的。所有范围构件共用一套公用的图形元素，每一个都有自己的 XWindow，接受自己的事件。它们都包含一个“滑槽”和一个“滑块”。用鼠标指针拖动滑块可以在滑槽中前后移动，在滑块前后的滑槽中点击，滑块就会前后大步移动。

和前面提到的调整对象一样，所有范围构件是与一个调整对象相关联的。该对象会计算滑块的长度和在滑槽中的位置。当用户操纵滑块时，范围构件会改变调整值。

8.1 滚动条构件GtkScrollBar

这是标准的滚动条。一般只用于一些需要滚动条的构件，比如列表、文本构件，或视角构件(在很多情况下使用滚动窗口构件更方便)。对其他目的,应该使用比例构件，因为它更友好，而且有更多的特性。

有水平和垂直滚动条两种类型。可以用下面的函数创建滚动条：

```
GtkWidget *gtk_hscrollbar_new( GtkAdjustment *adjustment);  
GtkWidget *gtk_vscrollbar_new( GtkAdjustment *adjustment);
```

adjustment参数可以是一个指向已有调整对象的指针或 NULL，当为NULL时会自动创建一个。如果希望将新的调整值传递给其他构件，例如文本构件的构造函数，在这种情况下指定NULL是很有用的。

8.2 比例构件GtkScale

GtkScale(比例构件)一般用于允许用户在一个指定的取值范围内可视地选择和操纵一个值。例如，在图片的缩放预览中调整放大倍数，或控制一种颜色的亮度，或在指定屏幕保护启动之前不活动的时间间隔时，可能需要用到比例构件。

有两种不同类型的比例构件：水平的和垂直的比例构件。大多数程序员似乎喜欢水平的比例构件。既然在本质上它们的工作方法是相同的，那么不需要对它们分别对待。

用下面的函数创建水平和垂直的比例构件：

```
GtkWidget *gtk_vscale_new( GtkAdjustment *adjustment);  
GtkWidget *gtk_hscale_new( GtkAdjustment *adjustment);
```

adjustment参数可以是一个已经用 gtk_adjustment_new() 创建的调整对象或 NULL，此时,会创建一个匿名的调整对象，所有的值都设为 0.0(在此处用处不大)。

为了避免引起困惑，可能要创建一个 page_size 的值设为 0.0 的调整对象，让它的实际上限值与用户能选择的最高值相对应。

8.2.1 函数和信号

比例构件可以在滑槽的旁边以数字形式显示其当前值。默认行为是显示值，但是可以用下

面的函数改变其行为：

```
void gtk_scale_set_draw_value( GtkScale *scale, gint draw_value );
```

可以猜到，draw_value取值为TRUE或FALSE，结果是显示或不显示。

缺省情况下，比例构件显示的值，也就是在 GtkAdjustment定义中的value域，圆整到一位小数。可以用以下函数改变显示的小数位：

```
void gtk_scale_set_digits( GtkScale *scale, gint digits );
```

digits是要显示的小数位数。可以设置为任意位数，但是实际上屏幕上最多只能显示 13位小数。

最后，显示的值可以放在滑槽的不同位置：

```
void gtk_scale_set_value_pos( GtkScale *scale, GtkPositionType pos );
```

参数pos是GtkPositionType类型，<gtk/gtkenums.h>中有它的定义，可以取以下值之一：

```
GTK_POS_LEFT GTK_POS_RIGHT
```

```
GTK_POS_TOP GTK_POS_BOTTOM
```

如果将值显示在滑槽的“边上”（例如，在水平比例构件的滑槽的顶部和底部），显示的值将跟随滑块上下移动。

8.2.2 常用的范围函数

范围构件本质上来说都是相当复杂的，不过，像所有“基本类”构件一样，绝大部分函数只有当你想彻底了解它们时才有用。另外，几乎所有函数和信号只在用它们写派生构件时才管用。但是，在<gtk/gtkrange.h>中还是有一些很有用的函数，并且在所有范围构件都起作用。

1. 设置更新方式

范围构件的“更新方式”定义了用户与构件交互时的调整值如何变化，以及调整值如何引发value_changed信号。更新方式在<gtk/gtkenums.h>中定义为GtkUpdateType枚举类型，有以下取值：

- GTK_UPDATE_POLICY_CONTINUOUS：这是默认值。value_changed信号是连续引发，例如，每当滑块移动，甚至移动最小数量时都会引发。
- GTK_UPDATE_POLICY_DISCONTINUOUS：只有滑块停止移动，用户释放鼠标键时才引发value_changed信号。
- GTK_UPDATE_POLICY_DELAYED：当用户释放鼠标键，或者滑块短期停止移动时才引发value_canged信号。

范围构件的更新方式可以用以下方法设置：用 GTK_RANGE(Widget)宏将变量转换为构件指针，并将它传递给以下函数：

```
void gtk_range_set_update_policy( GtkRange *range, GtkUpdateType policy );
```

2. 获得和设置调整值

可以用以下函数快速设置或取得调整值：

```
GtkAdjustment* gtk_range_get_adjustment( GtkRange *range );
```

```
void gtk_range_set_adjustment( GtkRange *range, GtkAdjustment *adjustment);
```

gtk_range_get_adjustment()：返回一个指向范围构件所连接的调整对象的指针。

gtk_range_set_adjustment()：如果将一个范围构件已经连接的调整传递到函数里面，什么也不会发生，不管是否改变其内部的值。

如果将新的调整对象传递进去, 如果存在旧的调整对象, 它会解除旧连接(可能会销毁它), 将适当的信号连接到新的调整对象, 并且调用私有函数 `gtk_range_adjustment_changed()`, 该函数将(或至少假装会)重新计算滑块的位置和尺寸, 并重新绘出该构件。

正如在调整对象部分所提到的, 如果想重新使用同一个调整对象, 当直接修改它的值时, 应该让它引发一个“changed”信号, 如下所示:

```
gtk_signal_emit_by_name (GTK_OBJECT (adjustment), "changed");
```

8.2.3 键盘和鼠标绑定

所有的GtkRange在鼠标点击交互时方式的多少是相同的。在滑槽上单击鼠标左键使调整值加上或减去一个 `page_increment`值, 滑块也移动相应的距离。在滑槽上单击鼠标右键将使滑块跳到鼠标点击处。在滚动条的箭头会使它的调整值改变一个 `step_increment`值。

要习惯使用它可能会花一点时间。不过, 在 GTK中, 缺省情况下的滚动条和比例构件可以获得键盘焦点。如果用户会感到困惑, 可以在滚动条的以下函数中用 `GTK_CAN_FOCUS`标志禁用:

```
GTK_WIDGET_UNSET_FLAGS (scrollbar, GTK_CAN_FOCUS);
```

按键绑定(当然只在该构件获得焦点时有效)在水平和垂直范围构件上略有不同, 理由很明显。但这些对比例构件和滚动条构件来说却不尽相同, 原因也不太明显(对滚动窗口中的水平和垂直滚动条来说, 可以避免这种困惑, 在这种情况下, 它们都对同一个区域操作)。

1. 垂直范围构件

所有垂直范围构件能用向上和向下键操作, 还可以用 `PageUp` 和 `PageDown`操作。上下箭头使滑块以 `step_increment`量上下移动, `PageUp`和 `PageDown`使滑块以 `page_increment`量上下移动。用户可以使用键盘让滑块在滑槽的两端之间自由移动。

对垂直比例构件, 移动滑块是用 `Home`和 `End`键, 然而对垂直滚动条构件, 用 `Ctrl+PageUp`和 `Ctrl+PageDown`操纵滑块。

2. 水平范围构件

对水平范围构件, 向左和向右键与垂直范围构件的向上和向下键起同样的作用。 `Home`和 `End`键将滑块移动到滑槽的头部和尾部。

对水平比例构件, 用 `Ctrl+向左键`以及 `Ctrl+向右键`使滑块以 `page_increment`量移动滑块, 对水平滚动条构件, 用 `Ctrl+Home`和 `Ctrl+End`做同样的工作。

8.2.4 示例

在一个窗口上放置了三个范围构件, 都连接到同一个调整对象, 并使用上面提到的一些调整范围构件参数的控制方法, 这样可以看到这些构件的效果。

```
/* GtkRange示例开始rangewidgets.c */

#include <gtk/gtk.h>
GtkWidget *hscale, *vscale;
void cb_pos_menu_select( GtkWidget      *item,
                        GtkPositionType pos )
{
    /* 设置比例构件的值 */
}
```

```
gtk_scale_set_value_pos (GTK_SCALE (hscale), pos);
gtk_scale_set_value_pos (GTK_SCALE (vscale), pos);
}

void cb_update_menu_select( GtkWidget      *item,
                           GtkUpdateType  policy )
{
    /* 设置比例构件的更新方式 */
    gtk_range_set_update_policy (GTK_RANGE (hscale), policy);
    gtk_range_set_update_policy (GTK_RANGE (vscale), policy);
}

void cb_digits_scale( GtkAdjustment *adj )
{
    /* 设置adj->value圆整的小数位数 */
    gtk_scale_set_digits (GTK_SCALE (hscale), (gint) adj->value);
    gtk_scale_set_digits (GTK_SCALE (vscale), (gint) adj->value);
}

void cb_page_size( GtkAdjustment *get,
                  GtkAdjustment *set )
{
    /* 将示例调整对象的page_size和page_increment值设置
     * 为"Page Size指定的值 */
    set->page_size = get->value;
    set->page_increment = get->value;

    /* 现在, 引发一个"changed"信号, 以重新配置所有
     * 已经连接到这个调整对象的构件 */
    gtk_signal_emit_by_name (GTK_OBJECT (set), "changed");
}

void cb_draw_value( GtkToggleButton *button )
{
    /* 根据开关按钮的状态设置在比例构件上是否显示比例值 */
    gtk_scale_set_draw_value (GTK_SCALE (hscale), button->active);
    gtk_scale_set_draw_value (GTK_SCALE (vscale), button->active);
}

GtkWidget *make_menu_item( gchar      *name,
                           GtkSignalFunc  callback,
                           gpointer      data )
{
    GtkWidget *item;

    item = gtk_menu_item_new_with_label (name);
    gtk_signal_connect (GTK_OBJECT (item), "activate",
                       callback, data);
    gtk_widget_show (item);
    return(item);
}
```



```

void scale_set_default_values( GtkScale *scale )
{
    gtk_range_set_update_policy (GTK_RANGE (scale),
                                GTK_UPDATE_CONTINUOUS);

    gtk_scale_set_digits (scale, 1);
    gtk_scale_set_value_pos (scale, GTK_POS_TOP);
    gtk_scale_set_draw_value (scale, TRUE);
}

/* 创建示例窗口 */

void create_range_controls( void )
{
    GtkWidget *window;
    GtkWidget *box1, *box2, *box3;
    GtkWidget *button;
    GtkWidget *scrollbar;
    GtkWidget *separator;
    GtkWidget *opt, *menu, *item;
    GtkWidget *label;
    GtkWidget *scale;
    GObject *adj1, *adj2;

    /* 标准的创建窗口代码 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC(gtk_main_quit),
                        NULL);

    gtk_window_set_title (GTK_WINDOW (window), "range controls");

    box1 = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), box1);
    gtk_widget_show (box1);

    box2 = gtk_hbox_new (FALSE, 10);
    gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
    gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
    gtk_widget_show (box2);

    /* 注意, page_size只对滚动条条件有区别, 并且, 实际上取得的最高值
    * 就是(upper - page_size) */
    adj1 = gtk_adjustment_new (0.0, 0.0, 101.0, 0.1, 1.0, 1.0);

    vscale = gtk_vscale_new (GTK_ADJUSTMENT (adj1));
    scale_set_default_values (GTK_SCALE (vscale));
    gtk_box_pack_start (GTK_BOX (box2), vscale, TRUE, TRUE, 0);
    gtk_widget_show (vscale);

    box3 = gtk_vbox_new (FALSE, 10);

```

```
gtk_box_pack_start (GTK_BOX (box2), box3, TRUE, TRUE, 0);
gtk_widget_show (box3);

/* 重新使用同一个调整对象 */
hscale = gtk_hscale_new (GTK_ADJUSTMENT (adj1));
gtk_widget_set_usize (GTK_WIDGET (hscale), 200, 30);
scale_set_default_values (GTK_SCALE (hscale));
gtk_box_pack_start (GTK_BOX (box3), hscale, TRUE, TRUE, 0);
gtk_widget_show (hscale);

/* 再次重用同一个调整对象 */
scrollbar = gtk_hscrollbar_new (GTK_ADJUSTMENT (adj1));
/* 注意, 这导致当滚动条移动时, 比例构件总是连续更新 */
gtk_range_set_update_policy (GTK_RANGE (scrollbar),
                             GTK_UPDATE_CONTINUOUS);
gtk_box_pack_start (GTK_BOX (box3), scrollbar, TRUE, TRUE, 0);
gtk_widget_show (scrollbar);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

/* 用一个检查按钮控制是否显示比例构件的值 */
button = gtk_check_button_new_with_label ("Display value on scale
widgets");
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);
gtk_signal_connect (GTK_OBJECT (button), "toggled",
                   GTK_SIGNAL_FUNC(cb_draw_value), NULL);
gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
gtk_widget_show (button);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);

/* 用一个选项菜单以改变显示值的位置 */
label = gtk_label_new ("Scale Value Position:");
gtk_box_pack_start (GTK_BOX (box2), label, FALSE, FALSE, 0);
gtk_widget_show (label);

opt = gtk_option_menu_new();
menu = gtk_menu_new();
item = make_menu_item ("Top",
                      GTK_SIGNAL_FUNC(cb_pos_menu_select),
                      GINT_TO_POINTER (GTK_POS_TOP));
gtk_menu_append (GTK_MENU (menu), item);
item = make_menu_item ("Bottom", GTK_SIGNAL_FUNC (cb_pos_menu_select),
                      GINT_TO_POINTER (GTK_POS_BOTTOM));
gtk_menu_append (GTK_MENU (menu), item);
item = make_menu_item ("Left", GTK_SIGNAL_FUNC (cb_pos_menu_select),
                      GINT_TO_POINTER (GTK_POS_LEFT));
```

```
gtk_menu_append (GTK_MENU (menu), item);
item = make_menu_item ("Right", GTK_SIGNAL_FUNC (cb_pos_menu_select),
    GINT_TO_POINTER (GTK_POS_RIGHT));
gtk_menu_append (GTK_MENU (menu), item);
gtk_option_menu_set_menu (GTK_OPTION_MENU (opt), menu);
gtk_box_pack_start (GTK_BOX (box2), opt, TRUE, TRUE, 0);
gtk_widget_show (opt);

gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);

/* 另一个选项菜单，这里是用于设置比例构件的更新方式 */
label = gtk_label_new ("Scale Update Policy:");
gtk_box_pack_start (GTK_BOX (box2), label, FALSE, FALSE, 0);
gtk_widget_show (label);
opt = gtk_option_menu_new();
menu = gtk_menu_new();

item = make_menu_item ("Continuous",
    GTK_SIGNAL_FUNC (cb_update_menu_select),
    GINT_TO_POINTER (GTK_UPDATE_CONTINUOUS));
gtk_menu_append (GTK_MENU (menu), item);

item = make_menu_item ("Discontinuous",
    GTK_SIGNAL_FUNC (cb_update_menu_select),
    GINT_TO_POINTER (GTK_UPDATE_DISCONTINUOUS));
gtk_menu_append (GTK_MENU (menu), item);
item = make_menu_item ("Delayed",
    GTK_SIGNAL_FUNC (cb_update_menu_select),
    GINT_TO_POINTER (GTK_UPDATE_DELAYED));
gtk_menu_append (GTK_MENU (menu), item);

gtk_option_menu_set_menu (GTK_OPTION_MENU (opt), menu);
gtk_box_pack_start (GTK_BOX (box2), opt, TRUE, TRUE, 0);
gtk_widget_show (opt);

gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);

/* 一个GtkHScale构件，用于调整示例比例构件的显示小数位数 */
label = gtk_label_new ("Scale Digits:");
gtk_box_pack_start (GTK_BOX (box2), label, FALSE, FALSE, 0);
gtk_widget_show (label);
adj2 = gtk_adjustment_new (1.0, 0.0, 5.0, 1.0, 1.0, 0.0);
gtk_signal_connect (GTK_OBJECT (adj2), "value_changed",
```

```

        GTK_SIGNAL_FUNC (cb_digits_scale), NULL);
scale = gtk_hscale_new (GTK_ADJUSTMENT (adj2));
gtk_scale_set_digits (GTK_SCALE (scale), 0);
gtk_box_pack_start (GTK_BOX (box2), scale, TRUE, TRUE, 0);
gtk_widget_show (scale);
gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
/* 最后一个水平比例构件用于调整滚动条的 page_size值 */
label = gtk_label_new ("Scrollbar Page Size:");
gtk_box_pack_start (GTK_BOX (box2), label, FALSE, FALSE, 0);
gtk_widget_show (label);
adj2 = gtk_adjustment_new (1.0, 1.0, 101.0, 1.0, 1.0, 0.0);
gtk_signal_connect (GTK_OBJECT (adj2), "value_changed",
        GTK_SIGNAL_FUNC (cb_page_size), adj1);
scale = gtk_hscale_new (GTK_ADJUSTMENT (adj2));
gtk_scale_set_digits (GTK_SCALE (scale), 0);
gtk_box_pack_start (GTK_BOX (box2), scale, TRUE, TRUE, 0);
gtk_widget_show (scale);
gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);
separator = gtk_hseparator_new ();
gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 0);
gtk_widget_show (separator);

box2 = gtk_vbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, TRUE, 0);

gtk_widget_show (box2);
button = gtk_button_new_with_label ("Quit");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
        GTK_SIGNAL_FUNC(gtk_main_quit),
        NULL);
gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
gtk_widget_grab_default (button);
gtk_widget_show (button);
gtk_widget_show (window);
}

int main( int    argc,
        char *argv[] )
{
    gtk_init(&argc, &argv);
    create_range_controls();
    gtk_main();
    return(0);
}
/* 示例结束 */

```

可以注意到程序没有对 `delete_event` 事件调用 `gtk_signal_connect` 函数，仅仅对 `destroy` 信号调用了该函数。但是 `destroy` 函数一样会执行，因为对给定窗口来说，未经处理的 `delete_event` 事件会引发一个 `destroy` 信号。

将上面的示例代码保存为 `rangewidget.c`，然后写一个如下所示的 Makefile 文件：

```
CC = gcc
rangewidgets: rangewidgets.c
    $(CC) `gtk-config --cflags` rangewidgets.c -o \
        rangewidgets `gtk-config --libs`
```

```
clean:
    rm -f *.o rangewidgets
```

编译后，在 shell 提示符下输入 `./rangewidget` 执行该程序，运行效果见图 8-1。

可以用鼠标指针调整对象的取值。点击 `Display value on scale widgets` 检查按钮可以决定是否显示比例构件的值；在选项菜单 `Scale Value Position` 中选择合适的选项可以设置在何处显示比例构件的值；在选项菜单 `Scale Update Policy` 中选择合适的值用来设置比例菜单的更新方式；滑动 `Scale Digits` 可以设置比例构件的显示值的小数位；滑动 `Scrollbar Page Size` 设置滚动条构件的 `page_size` 值。点击 `Quit` 按钮退出应用程序。

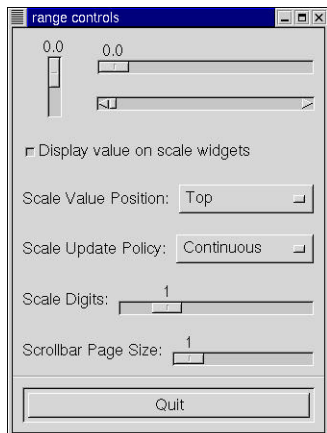


图8-1 范围构件

第9章 杂项构件

9.1 标签构件GtkLabel

GtkLabel(标签构件)是GTK中最常用的构件，实际上它很简单。因为没有相关联的 X窗口，标签构件不能引发信号。如果需要引发信号，可以将它放在一个事件盒构件中，或放在按钮构件里面。

用以下函数创建新标签构件：

```
GtkWidget *gtk_label_new(char *str );
```

唯一的参数是要由标签显示的字符串。

创建标签构件后，要改变标签内的文本，用以下函数：

```
void gtk_label_set_text( GtkLabel *label,char *str );
```

第一参数是前面创建的标签构件(用GTK_LABEL()宏转换)，并且第二个参数是新字符串。如果需要，新字符串需要的空间会做自动调整。在字符串中放置换行符，可以创建多行标签。

用以下函数取得标签的当前文本：

```
void gtk_label_get( GtkLabel *Label,char **str );
```

第一个参数是前面创建的标签构件，并且第二个参数是要返回的字符串。不要释放返回的字符串，因为GTK内部要使用它。

标签的文本可以用以下函数设置对齐方式：

```
void gtk_label_set_justify( GtkLabel *Label,  
                           GtkJustification jtype );
```

jtype的值可以是：

GTK_JUSTIFY_LEFT	左对齐
GTK_JUSTIFY_RIGHT	右对齐
GTK_JUSTIFY_CENTER	居中对齐(默认)
GTK_JUSTIFY_FILL	充满

标签构件的文本会自动换行。用以下函数激活“自动换行”：

```
void gtk_label_set_line_wrap (GtkLabel *Label, gboolean wrap);
```

wrap参数可取TRUE或FALSE，对应于自动换行和自动不自动换行。

如果想要使标签构件加下划线，可以在标签构件中设置显示模式：

```
void gtk_label_set_pattern (GtkLabel *Label,const gchar *pattern);
```

pattern参数指定下划线的外观。它由一串下划线和空格组成。下划线指示标签的相应字符应该加一个下划线。例如，“__ ”将在标签的第1、第2个字符和第8、第9个字符加下划线。

下面是一个说明这些函数的短例子。这个例子用框架构件能更好地示范标签的风格。

```
/* GtkLabel示例开始 label.c */
```

```
#include <gtk/gtk.h>
```



```
int main( int    argc,
          char *argv[] )
{
    static GtkWidget *window = NULL;
    GtkWidget *hbox;
    GtkWidget *vbox;
    GtkWidget *frame;
    GtkWidget *label;

    /* 初始化GTK */
    gtk_init(&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC(gtk_main_quit),
                        NULL);

    gtk_window_set_title (GTK_WINDOW (window), "Label");
    vbox = gtk_vbox_new (FALSE, 5);
    hbox = gtk_hbox_new (FALSE, 5);
    gtk_container_add (GTK_CONTAINER (window), hbox);
    gtk_box_pack_start (GTK_BOX (hbox), vbox, FALSE, FALSE, 0);
    gtk_container_set_border_width (GTK_CONTAINER (window), 5);

    frame = gtk_frame_new ("Normal Label");
    label = gtk_label_new ("This is a Normal label");
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

    frame = gtk_frame_new ("Multi-line Label");
    label = gtk_label_new ("This is a Multi-line label.\nSecond line\n" \
                           "Third line");
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

    frame = gtk_frame_new ("Left Justified Label");
    label = gtk_label_new ("This is a Left-Justified\n" \
                           "Multi-line label.\nThird          line");
    gtk_label_set_justify (GTK_LABEL (label), GTK_JUSTIFY_LEFT);
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

    frame = gtk_frame_new ("Right Justified Label");
    label = gtk_label_new ("This is a Right-Justified\nMulti-line label.\n" \
                           "Fourth line, (j/k)");
    gtk_label_set_justify (GTK_LABEL (label), GTK_JUSTIFY_RIGHT);
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

    vbox = gtk_vbox_new (FALSE, 5);
    gtk_box_pack_start (GTK_BOX (hbox), vbox, FALSE, FALSE, 0);
```

```

frame = gtk_frame_new ("Line wrapped label");
label = gtk_label_new ("This is an example of a line-wrapped label.  It " \
    "should not be taking up the entire " \
    /* 一大段空格, 用来测试间距 */ \
    "width allocated to it, but automatically " \
    "wraps the words to fit.  " \
    "The time has come, for all good men, to come to " \
    "the aid of their party.  " \
    "The sixth sheik's six sheep's sick.\n" \
    "    It supports multiple paragraphs correctly, " \
    "and correctly adds "\
    "many        extra spaces. ");
gtk_label_set_line_wrap (GTK_LABEL (label), TRUE);
gtk_container_add (GTK_CONTAINER (frame), label);
gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

frame = gtk_frame_new ("Filled, wrapped label");
label = gtk_label_new ("This is an example of a line-wrapped, filled label
" \
    "It should be taking "\
    "up the entire        width allocated to it.  "

    "Here is a sentence to prove "\
    "my point.  Here is another sentence. "\
    "Here comes the sun, do de do de do.\n"\
    "    This is a new paragraph.\n"\
    "    This is another newer, longer, better " \
    "paragraph.  It is coming to an end, "\
    "unfortunately.");
gtk_label_set_justify (GTK_LABEL (label), GTK_JUSTIFY_FILL);
gtk_label_set_line_wrap (GTK_LABEL (label), TRUE);
gtk_container_add (GTK_CONTAINER (frame), label);
gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

frame = gtk_frame_new ("Underlined label");
label = gtk_label_new ("This label is underlined!\n"
    "This one is underlined in quite a funky fashion");
gtk_label_set_justify (GTK_LABEL (label), GTK_JUSTIFY_LEFT);
gtk_label_set_pattern (GTK_LABEL (label),
    "----- - ----- - \
    -----  -- -----  ");
gtk_container_add (GTK_CONTAINER (frame), label);
gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

gtk_widget_show_all (window);

gtk_main ();

return(0);
}
/* 示例结束 */

```

图9-1是上面代码的运行结果。这个例子展示了 GtkLabel构件的各种属性。

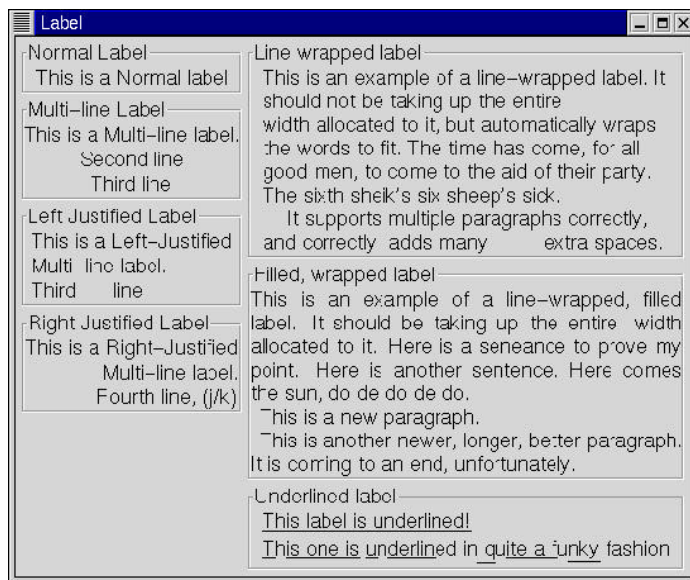


图9-1 标签构件

9.2 箭头构件GtkArrow

GtkArrow(箭头构件)画一个箭头，面向几种不同的方向，并有几种不同的风格。在许多应用程序中，常用于创建带箭头的按钮。和标签构件一样，它不能引发信号。

只有两个函数用来操纵箭头构件：

```
GtkWidget *gtk_arrow_new( GtkArrowType arrow_type,
                           GtkShadowType shadow_type );
void gtk_arrow_set( GtkArrow *arrow, GtkArrowType arrow_type,
                   GtkShadowType shadow_type );
```

第一个函数创建新的箭头构件，指明构件的类型和外观；第二个函数用来改变箭头构件类型和外观。

arrow_type参数指示箭头指向哪个方向，可以取下列值：

GTK_ARROW_UP	向上
GTK_ARROW_DOWN	向下
GTK_ARROW_LEFT	向左
GTK_ARROW_RIGHT	向右

shadow_type参数指明箭头的投影的类型，可以取下列值：

GTK_SHADOW_IN
GTK_SHADOW_OUT (缺省值)
GTK_SHADOW_ETCHED_IN
GTK_SHADOW_ETCHED_OUT

下面是说明这些类型和外观的例子。

```
/* GtkArrow示例arrow.c */
```

```
#include <gtk/gtk.h>

/* 用指定的参数创建一个箭头构件并将它组装到按钮中 */
GtkWidget *create_arrow_button( GtkArrowType  arrow_type,
                                GtkShadowType shadow_type )
{
    GtkWidget *button;
    GtkWidget *arrow;

    button = gtk_button_new();
    arrow = gtk_arrow_new (arrow_type, shadow_type);

    gtk_container_add (GTK_CONTAINER (button), arrow);

    gtk_widget_show(button);
    gtk_widget_show(arrow);

    return(button);
}

int main( int  argc,
          char *argv[] )
{
    /* 构件的存储类型是GtkWidget */
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box;

    /* 初始化Gtk */
    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title (GTK_WINDOW (window), "Arrow Buttons");

    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (gtk_main_quit), NULL);

    /* 设置窗口的边框的宽度 */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个组装盒以容纳箭头/按钮 */
    box = gtk_hbox_new (FALSE, 0);
    gtk_container_set_border_width (GTK_CONTAINER (box), 2);
    gtk_container_add (GTK_CONTAINER (window), box);

    /* 组装、显示所有的构件 */
    gtk_widget_show(box);

    button = create_arrow_button(GTK_ARROW_UP, GTK_SHADOW_IN);
```

```

gtk_box_pack_start (GTK_BOX (box), button, FALSE, FALSE, 3);

button = create_arrow_button(GTK_ARROW_DOWN, GTK_SHADOW_OUT);
gtk_box_pack_start (GTK_BOX (box), button, FALSE, FALSE, 3);

button = create_arrow_button(GTK_ARROW_LEFT, GTK_SHADOW_ETCHED_IN);
gtk_box_pack_start (GTK_BOX (box), button, FALSE, FALSE, 3);

button = create_arrow_button(GTK_ARROW_RIGHT, GTK_SHADOW_ETCHED_OUT);
gtk_box_pack_start (GTK_BOX (box), button, FALSE, FALSE, 3);

gtk_widget_show (window);

/* 进入主循环, 等待用户的动作 */
gtk_main ();

return(0);
}
/* 示例结束 */

```



图9-2 GtkArrow构件

上面代码的运行效果见图9-2。在窗口上创建了四个带箭头的按钮。

9.3 工具提示对象GtkTooltips

工具提示对象 (GtkTooltips) 就是当鼠标指针移到按钮或其他构件上并停留几秒时, 弹出的文本串。工具提示对象很容易使用, 所以在此仅仅对它们进行解释, 不再举例。本书的其他示例里面有很多都用到了工具提示对象。

不接收事件的构件 (没有自己的 X 窗口的构件) 不能和工具提示对象一起工作。

可以使用 `gtk_tooltips_new()` 函数创建工具提示对象。因为 GtkTooltips 对象可以重复使用, 一般在应用程序中仅需要调用这个函数一次。

```
GtkTooltips *gtk_tooltips_new(void);
```

一旦已创建新的工具提示, 并且希望在某个构件上应用它, 可调用以下函数设置它:

```

void gtk_tooltips_set_tip( GtkTooltips *tooltip,
GtkWidget *widget, const gchar *tip_text, const
gchar *tip_private );

```

第一个参数是已经创建的工具提示对象, 其后第二个参数是希望弹出工具提示的构件, 第三个参数是要弹出的文本。最后一个参数是作为标识符的文本串, 当用 GtkTipsQuery 实现上下文敏感的帮助时要引用该标识符。目前, 你可以把它设置为 NULL。

下面有个短例子:

```

GtkTooltips *tooltips;
GtkWidget *button;

.
.
tooltips = gtk_tooltips_new ();
button = gtk_button_new_with_label ("button 1");

.
.
gtk_tooltips_set_tip (tooltips, button, "This is button 1", NULL);

```

还有其他与工具提示有关的函数，下面仅仅列出一些函数的简要描述。

```
void gtk_tooltips_enable( GtkTooltips *tooltip);
```

激活已经禁用的工具提示对象。

```
void gtk_tooltips_disable( GtkTooltips *tooltip);
```

禁用已经激活的工具提示对象。

```
void gtk_tooltips_set_delay( GtkTooltips *tooltip, gint delay);
```

设置鼠标在构件上停留多少毫秒后弹出工具提示，默认是 500毫秒(半秒)。

```
void gtk_tooltips_set_colors( GtkTooltips *tooltips,
                             GdkColor      *background,
                             GdkColor      *foreground );
```

设置工具提示的前景色和背景色。

上面是所有与工具提示有关的函数，实际上比你想要知道的还多。

9.4 进度条构件GtkProgressBar

进度条用于显示正在进行的操作的状态。它相当容易使用，在下面的代码中可以看到。下面的内容从创建一个新进度条开始。

有两种方法创建进度条，简单的方法不需要参数，另一种方法用一个调整对象作为参数。如果前者使用,进度条创建它拥有的调整对象。

```
GtkWidget *gtk_progress_bar_new(void);
```

```
GtkWidget *gtk_progress_bar_new_with_adjustment( GtkAdjustment *adjustment);
```

第二种方法的优势是我们能用调整对象明确指定进度条的范围参数。

进度条的调整对象能用下面的函数动态改变：

```
void gtk_progress_set_adjustment( GtkProgress *progress,
                                  GtkAdjustment *adjustment);
```

既然进度条已经创建，那就可以使用它了。

```
void gtk_progress_bar_update( GtkProgressBar *pbar, gfloat percentage);
```

更新进度条时，第一个参数是希望操作的进度条，第二个参数是“已完成”的百分比，意思是进度条从0~100%已经填充的数量。它以0~1范围的实数传递给函数。

GTK 1.2版已经给进度条添加了一个新的功能，那就是允许它以不同的方法显示其值，并通知用户它的当前值和范围。

进度条可以用以下函数设置它的移动方向：

```
void gtk_progress_bar_set_orientation( GtkProgressBar *pbar,
                                       GtkProgressBarOrientation orientation);
```

orientation参数可以取下列值之一，以指示进度条的移动方向：

GTK_PROGRESS_LEFT_TO_RIGHT 从左向右

GTK_PROGRESS_RIGHT_TO_LEFT 从右向左

GTK_PROGRESS_BOTTOM_TO_TOP 从下向上

GTK_PROGRESS_TOP_TO_BOTTOM 从上向下

进度条可以以连续和间断的方式显示进度处理的数值。在连续的方式下，进度条每个值都会更新；在间断方式下，进度条以不连续的方式更新。更新的次数是可配置的。

进度条的式样可以用以下函数进行更新：

```
void gtk_progress_bar_set_bar_style( GtkProgressBar *pbar,  
                                     GtkProgressBarStyle style);
```

style参数取以下两种值：

GTK_PROGRESS_CONTINUOUS 连续更新

GTK_PROGRESS_DISCRETE 间断更新

间断更新的次数可以用以下函数设置：

```
gtk_progress_bar_set_discrete_blocks( GtkProgressBar *pbar,  
                                       guint blocks);
```

除了指示进度已经发生的数量以外，进度条还可以设置为仅仅指示有活动在继续。这些设置在进度无法按数值度量的情况下很有用。进度条的活动模式不会受进度条的 style 参数的影响，并且会覆盖它的设置。其方式只能是 TRUE 或者 FALSE，可以用下列函数确定：

```
void gtk_progress_set_activity_mode( GtkProgress *progress,  
                                     guint activity_mode );
```

活动指示的步数和活动的块数由以下函数设置：

```
void gtk_progress_bar_set_activity_step( GtkProgressBar *pbar, guint step);  
void gtk_progress_bar_set_activity_blocks( GtkProgressBar *pbar, guint  
blocks);
```

在连续的方式下，进度条可以用下列函数在滑槽内显示一个可配置的文本串：

```
void gtk_progress_set_format_string( GtkProgress *progress, gchar *format);
```

其中，format 参数与 C 语言中的 printf 函数的格式参数相似。

下列格式可以用于格式化串：

%p 百分比

%v 值

%l 低范围值

%u 高限范围值

是否显示文本串可以用以下函数开/关：

```
void gtk_progress_set_show_text( GtkProgress *progress, gint show_text );
```

show_text 参数是布尔型的，可取值为 TRUE/FALSE。

文本的外观能做进一步的修改，如下所示：

```
void gtk_progress_set_text_alignment( GtkProgress *progress,  
gfloat x_align, gfloat y_align );
```

x_align 和 y_align 参数取 0.0 和 1.0 之间的值。这些值指定文本串在滑槽内的位置。两个值都取为 0.0 将把文本串放在左上角；取值 0.5 (默认) 让文本居于滑槽中心；设为 1.0 则将文本串置于右下角。

进度条对象的当前文本设置能由下列两个函数从当前文本或从指定的调整对象中取得。这些函数返回的字符串应该由应用程序释放 (用 g_free() 函数)。这些函数返回要显示在滑槽内的格式化字符串。

```
gchar *gtk_progress_get_current_text( GtkProgress *progress );  
gchar *gtk_progress_get_text_from_value( GtkProgress *progress, gfloat value );
```

还有另一种改变进度条对象的范围和取值的方法。使用下列函数：

```
void gtk_progress_configure( GtkProgress *progress, gfloat value, gfloat min,  
gfloat max);
```

这个函数提供了相当简单的使用进度条对象的范围和取值的接口。

其余的函数可以以各种类型和格式获得或设置当前进度条对象的值：

```
void gtk_progress_set_percentage( GtkProgress *progress, gfloat percentage);
void gtk_progress_set_value( GtkProgress *progress, gfloat value);
gfloat gtk_progress_get_value( GtkProgress *progress);
gfloat gtk_progress_get_current_percentage( GtkProgress *progress);
gfloat gtk_progress_get_percentage_from_value( GtkProgress *progress,
gfloat value);
```

这些函数从字面上就可以了解它的含义。最后一个函数使用指定进度条的调整对象计算给定范围值的百分比。

进度条通常和timeout或其他函数同时使用，使应用程序就像是多任务一样。一般都用同样的方式使用gtk_progress_bar_update函数。

下面是一个进度条的例子，用 timeout函数更新进度条的值。代码也演示了怎样复位进度条。

```
/* GtkProgressBar示例progressbar.c */

#include <gtk/gtk.h>

typedef struct _ProgressData {
    GtkWidget *window;
    GtkWidget *pbar;
    int timer;
} ProgressData;

/* 更新进度条，这样就能够看到进度条的移动 */
gint progress_timeout( gpointer data )
{
    gfloat new_val;
    GtkAdjustment *adj;

    /* 使用在调整对象中设置的取值范围计算进度条的值 */

    new_val = gtk_progress_get_value( GTK_PROGRESS(data) ) + 1;

    adj = GTK_PROGRESS (data)->adjustment;
    if (new_val > adj->upper)
        new_val = adj->lower;

    /* 设置进度条的新值 */
    gtk_progress_set_value (GTK_PROGRESS (data), new_val);

    /* 这是一个timeout函数，返回TRUE，这样它就能够继续调用 */
    return(TRUE);
}

/* 回调函数，切换在进度条内的滑槽上的文本显示 */
void toggle_show_text( GtkWidget *widget,
    ProgressData *pdata )
```



```
{
    gtk_progress_set_show_text (GTK_PROGRESS (pdata->pbar),
                                GTK_TOGGLE_BUTTON (widget)->active);
}

/* 回调函数，切换进度条的活动模式 */
void toggle_activity_mode( GtkWidget      *widget,
                           ProgressData *pdata )
{
    gtk_progress_set_activity_mode (GTK_PROGRESS (pdata->pbar),
                                    GTK_TOGGLE_BUTTON (widget)->active);
}

/* 回调函数，切换进度条的连续模式 */
void set_continuous_mode( GtkWidget      *widget,
                          ProgressData *pdata )
{
    gtk_progress_bar_set_bar_style (GTK_PROGRESS_BAR (pdata->pbar),
                                    GTK_PROGRESS_CONTINUOUS);
}

/* 回调函数，切换进度条的间断模式 */
void set_discrete_mode( GtkWidget      *widget,
                        ProgressData *pdata )
{
    gtk_progress_bar_set_bar_style (GTK_PROGRESS_BAR (pdata->pbar),
                                    GTK_PROGRESS_DISCRETE);
}

/* 清除分配的内存，删除timeout函数 */
void destroy_progress( GtkWidget      *widget,
                      ProgressData *pdata )
{
    gtk_timeout_remove (pdata->timer);
    pdata->timer = 0;
    pdata->window = NULL;
    g_free(pdata);
    gtk_main_quit();
}

int main( int    argc,
          char *argv[])
{
    ProgressData *pdata;
    GtkWidget *align;
    GtkWidget *separator;
    GtkWidget *table;
    GtkAdjustment *adj;
    GtkWidget *button;
    GtkWidget *check;
    GtkWidget *vbox;
```

```
gtk_init (&argc, &argv);

/* 为传递到回调函数中的数据分配内存 */
pdata = g_malloc( sizeof(ProgressData) );

pdata->window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_policy (GTK_WINDOW (pdata->window), FALSE, FALSE, TRUE);

gtk_signal_connect (GTK_OBJECT (pdata->window), "destroy",
                    GTK_SIGNAL_FUNC (destroy_progress),
                    pdata);

gtk_window_set_title (GTK_WINDOW (pdata->window), "GtkProgressBar");
gtk_container_set_border_width (GTK_CONTAINER (pdata->window), 0);

vbox = gtk_vbox_new (FALSE, 5);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);
gtk_container_add (GTK_CONTAINER (pdata->window), vbox);
gtk_widget_show(vbox);

/* 创建一个居中对齐的对象 */
align = gtk_alignment_new (0.5, 0.5, 0, 0);
gtk_box_pack_start (GTK_BOX (vbox), align, FALSE, FALSE, 5);
gtk_widget_show(align);

/* 创建一个调整对象，以保持进度条的范围值 */
adj = (GtkAdjustment *) gtk_adjustment_new (0, 1, 150, 0, 0, 0);

/* 使用前面创建的调整对象创建一个进度条 */
pdata->pbar = gtk_progress_bar_new_with_adjustment (adj);

/* 设置能显示在进度条的滑槽内的字符串的格式
 * %p - 百分比
 * %v - 值
 * %l - 底限范围值
 * %u - 上限范围值 */
gtk_progress_set_format_string (GTK_PROGRESS (pdata->pbar),
                                "%v from [%l-%u] (= %p%)");
gtk_container_add (GTK_CONTAINER (align), pdata->pbar);
gtk_widget_show(pdata->pbar);

/* 添加一个计时的回调函数，以更新进度条的值 */
pdata->timer = gtk_timeout_add (100, progress_timeout, pdata->pbar);

separator = gtk_hseparator_new ();
gtk_box_pack_start (GTK_BOX (vbox), separator, FALSE, FALSE, 0);
gtk_widget_show(separator);

/* 行数、列数、同质性 */
table = gtk_table_new (2, 3, FALSE);
gtk_box_pack_start (GTK_BOX (vbox), table, FALSE, TRUE, 0);
gtk_widget_show(table);
```

```
/* 添加一个检查按钮，以选择显示在滑槽内的文本 */
check = gtk_check_button_new_with_label ("Show text");
gtk_table_attach (GTK_TABLE (table), check, 0, 1, 0, 1,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
gtk_signal_connect (GTK_OBJECT (check), "clicked",
                   GTK_SIGNAL_FUNC (toggle_show_text),
                   pdata);
gtk_widget_show(check);

/* 添加一个检查按钮，切换活动状态 */
check = gtk_check_button_new_with_label ("Activity mode");
gtk_table_attach (GTK_TABLE (table), check, 0, 1, 1, 2,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
gtk_signal_connect (GTK_OBJECT (check), "clicked",
                   GTK_SIGNAL_FUNC (toggle_activity_mode),
                   pdata);
gtk_widget_show(check);

separator = gtk_vseparator_new ();
gtk_table_attach (GTK_TABLE (table), separator, 1, 2, 0, 2,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
gtk_widget_show(separator);

/* 添加一个无线按钮，以选择连续选择模式 */
button = gtk_radio_button_new_with_label (NULL, "Continuous");
gtk_table_attach (GTK_TABLE (table), button, 2, 3, 0, 1,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (set_continuous_mode),
                   pdata);
gtk_widget_show (button);

/* 添加一个无线按钮，以选择间断模式 */
button = gtk_radio_button_new_with_label(
    gtk_radio_button_group (GTK_RADIO_BUTTON (button)),
    "Discrete");
gtk_table_attach (GTK_TABLE (table), button, 2, 3, 1, 2,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (set_discrete_mode),
                   pdata);
gtk_widget_show (button);

separator = gtk_hseparator_new ();
gtk_box_pack_start (GTK_BOX (vbox), separator, FALSE, FALSE, 0);
gtk_widget_show(separator);
```

```

/* 添加一个按钮，用来退出应用程序 */
button = gtk_button_new_with_label ("close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           (GtkSignalFunc) gtk_widget_destroy,
                           GTK_OBJECT (pdata->window));
gtk_box_pack_start (GTK_BOX (vbox), button, FALSE, FALSE, 0);

/* 将按钮设置为缺省的 */
GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);

/* 将按钮冻结为缺省按钮，只要按回车键
 * 就相当于点击了这个按钮 */
gtk_widget_grab_default (button);
gtk_widget_show(button);

gtk_widget_show (pdata->window);

gtk_main ();
return(0);
} /* 示例结束 */

```

将上面的代码保存为 progressbar.c，编译后的运行效果见图9-3。其中演示了进度条的不同更新方式，你还可以设置是否显示文本，以及是否处于获得模式。

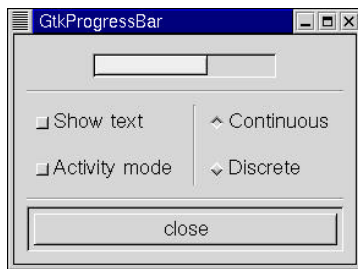


图9-3 进度条构件

9.5 对话框构件

对话框构件非常简单，事实上它仅仅是一个预先组装了几个构件到里面的窗口。

对话框是如下定义的：

```

struct GtkDialog { GtkWidget window;
GtkWidget *vbox; GtkWidget *action_area; };

```

从上面可以看到，对话框只是简单地创建一个窗口，并在顶部组装一个 GtkWidget，然后在 GtkWidget 中组装一个分隔线，再加一个称为“活动区”的 GtkWidget。

对话框构件可以用于弹出消息，或者其他类似的任务。它只有一个函数：

```
GtkWidget *gtk_dialog_new(void);
```

要创建对话框，可以使用如下所示的函数：

```

GtkWidget *window;
window = gtk_dialog_new ();

```

你还可以在活动区中组装一个按钮：

```

button = ...
gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area),
                    button, TRUE, TRUE, 0);
gtk_widget_show (button);

```

还可以在 vbox 里面组装一个标签构件，像下面那样：

```

label = gtk_label_new ("XXXXX");
gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->vbox), label,
                    TRUE, TRUE, 0);
gtk_widget_show (label);

```

作为一个例子，可以在对话框里面组装两个按钮：一个“确定”按钮和一个“取消”按钮，以便向用户提出疑问，或显示一个错误信息。然后可以把不同信号连接到每个按钮，对用户的选择进行响应。

如果由对话框提供的垂直和水平组装盒的简单功能不能满足你的需要，可以简单地在组装盒中添加其他外观构件。例如，可以在其中添加一个表格构件。

因为 `GtkDialog` 太简单，不能满足快速开发应用程序的要求，所以建议你使用 `GnomeDialog` 及其子构件。`GnomeDialog` 将在第15章中介绍。

9.6 pixmap

`pixmap` 是包含图像的数据结构。这些图像可以用于不同的地方，但是最常见的是 X 桌面的图标，或用作鼠标指针。

仅有2种颜色的 `pixmap` 称为位图，另外有一些例程用于处理这种情况。

了解 `pixmap` 有助于了解 X 窗口系统如何工作。在 X 系统下，与用户交互的应用程序不一定要在同一台计算机上运行。不同的应用程序，称为“客户”同时与一个显示图像和处理键盘和鼠标的程序通讯。这个直接与用户交互的程序被称为“显示服务器”或“X服务器”。

既然通信可能发生在网络上，那么，与 X 服务器保持一些信息是非常重要的。例如 `pixmap`，就是储存在 X 服务器的内存中的。这意味着一旦设置了 `pixmap` 的值，它们就不再需要通过网络传输，而是传送一个“在此处显示编号为 XYZ 的 `pixmap`”。即使你目前没有在 X 系统下使用 GTK，使用像 `pixmap` 这样的结构也会使应用程序在 X 下工作得更好。

`pixmap` 能从内存中的数据创建，或者用从文件中读出的数据创建。这里我们将介绍每种创建 `pixmap` 的方法。

```
GdkPixmap *gdk_bitmap_create_from_data( GdkWindow *window,  
gchar *data, gint width, gint height );
```

这个函数使用从内存读取的数据创建单个位平面的 `pixmap` (2种颜色)。数据的每位分别代表像素是打开或关闭。宽和高是以像素度量的。`GdkWindow` 指针指向当前窗口，因为 `pixmap` 的资源指的是它要显示的屏幕的场合。

```
GdkPixmap *gdk_pixmap_create_from_data( GdkWindow *window,  
gchar *data,  
gint width,  
gint height,  
gint depth,  
GdkColor *fg,  
GdkColor *bg );
```

这个函数用于从指定的位图数据创建给定深度（颜色数）的 `pixmap` 图片。`fg` 和 `bg` 是要用到的前景和背景色。

```
GdkPixmap *gdk_pixmap_create_from_xpm_d( GdkWindow *window,  
GdkBitmap **mask,  
GdkColor *transparent_color,  
gchar **data );
```

XPM 格式是 X 窗口系统中一种可读的 `pixmap` 图形表示，它得到了广泛的应用。有许多实用程序可以用来创建这种格式的图片。由 `filename` 参数指定的文件必须包含 XPM 格式的图形，这种图形会被加载到 `pixmap` 结构中。`Mask` 参数指定在一个 `pixmap` 中哪一位是不透明的。所有

的其他位都使用由transparent_color所指定的颜色着色。下面是一个例子：

```
GdkPixmap *gdk_pixmap_create_from_xpm_d( GdkWindow *window,
                                           GdkBitmap **mask,
                                           GdkColor *transparent_color,
                                           gchar **data );
```

小图像能以XPM格式合并到程序里面，一个 pixmap格式的图片能够用这些数据创建，而不是从文件中读进来。下面是这种数据的例子。

```

/* XPM */
static const char * xpm_data[] = {
    "16 16 3 1",
    "      c None",
    ".      c #0000000000000",
    "X      c #FFFFFFFFFFFF",
    "      ",
    "      . . . . .",
    "      .XXX.X.",
    "      .XXX.XX.",
    "      .XXX.XXX.",
    "      .XXX....",
    "      .XXXXXXXX.",
    "      .XXXXXXXX.",
    "      .XXXXXXXX.",
    "      .XXXXXXXX.",
    "      .XXXXXXXX.",
    "      .XXXXXXXX.",
    "      .XXXXXXXX.",
    "      .XXXXXXXX.",
    "      .XXXXXXXX.",
    "      . . . . .",
    "      ",
    "      ",
    "      "};

```

当已经用完pixmap，并且不太可能再次使用它时，最好用 gdk_pixmap_unref()函数将这些资源释放。pixmap应该被看作是宝贵的资源，因为它们占据了终端用户的 X服务器进程的内存。即使开发应用程序时所用的计算机是一台功能强劲的“服务器”，最终用户使用的却可能是速度较慢的PC，因而释放不再使用的资源是一件很重要的工作。

一旦我们创建了pixmap，就能将它作为Gtk构件显示。我们必须创建一个GtkPixmap构件以包含GDK pixmap图片。用下面的函数实现：

```
GtkWidget *gtk_pixmap_new( GdkPixmap *pixmap, GdkBitmap *mask );
```

其他的相关函数是：

```
guint gtk_pixmap_get_type(void);
void gtk_pixmap_set( GtkPixmap *pixmap, GdkPixmap *val, GdkBitmap *mask);
void gtk_pixmap_get( GtkPixmap *pixmap, GdkPixmap **val, GdkBitmap **mask);
```

gtk_pixmap_set函数一般用于正在管理的构件的 pixmap图片。val是用GDK创建的 pixmap图片。

下面的例子是在一个按钮上使用 pixmap 图片。

```
/* pixmap示例开始pixmap.c */
#include <gtk/gtk.h>
```

129

```

/* XPM数据，用于"打开文件"图标 */
static const char * xpm_data[] = {
    "16 16 3 1",
    "      c None",
    ".      c #000000000000",
    "X      c #FFFFFFFFFFFF",
    "",
    "      ",
    ".....",
    ".XXX.X.",
    ".XXX.XX.",
    ".XXX.XXX.",
    ".XXX.....",
    ".XXXXXXXX.",
    ".XXXXXXXX.",
    ".XXXXXXXX.",
    ".XXXXXXXX.",
    ".XXXXXXXX.",
    ".XXXXXXXX.",
    ".XXXXXXXX.",
    ".....",
    "",
    ""};

/* 调用这个函数时(通过delete_event信号)，终止应用程序*/
void close_application( GtkWidget *widget, GdkEvent *event, gpointer data ) {
    gtk_main_quit();
}

/*点击按钮时调用这个函数，并打印一条信息*/
void button_clicked( GtkWidget *widget, gpointer data ) {
    printf( "button clicked\n" );
}

int main( int argc, char *argv[] )
{
    /* 构件的存储格式是GtkWidget */
    GtkWidget *window, *pixmapwid, *button;
    GdkPixmap *pixmap;
    GdkBitmap *mask;
    GtkStyle *style;

    /* 创建主窗口，为delete_event信号设置回调函数以终止应用程序*/
    gtk_init( &argc, &argv );
    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    gtk_signal_connect( GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (close_application), NULL );
    gtk_container_set_border_width( GTK_CONTAINER (window), 10 );
    gtk_widget_show( window );

    /* 现在用gdk创建Pixmap */
    style = gtk_widget_get_style( window );

```

```

pixmap = gdk_pixmap_create_from_xpm_d( window->window, &mask,
                                         &style->bg[GTK_STATE_NORMAL],
                                         (gchar **)xpm_data );

```

```

/* 用一个Pixmap构件包含这个pixmap图片 */

```

```

pixmapwid = gtk_pixmap_new( pixmap, mask );
gtk_widget_show( pixmapwid );

```

```

/* 将pixmap构件放在一个按钮中 */

```

```

button = gtk_button_new();
gtk_container_add( GTK_CONTAINER(button), pixmapwid );
gtk_container_add( GTK_CONTAINER(window), button );
gtk_widget_show( button );

```

```

gtk_signal_connect( GTK_OBJECT(button), "clicked",
                    GTK_SIGNAL_FUNC(button_clicked), NULL );

```

```

/* 显示主窗口 */

```

```

gtk_main ();

```

```

return 0;

```

```

}

```

```

/*示例结束 */

```

如果要从当前目录的icon0.xpm中加载图片，我们可以用下面的方法创建 pixmap图片：

```

/* 从文件加载pixmap图片*/

```

```

pixmap = gdk_pixmap_create_from_xpm( window->window, &mask,
                                       &style->bg[GTK_STATE_NORMAL],
                                       " ./icon0.xpm" );

```

```

pixmapwid = gtk_pixmap_new( pixmap, mask );

```

```

gtk_widget_show( pixmapwid );

```

```

gtk_container_add( GTK_CONTAINER(window), pixmapwid );

```

使用pixmap图片的不利之处是它显示的对象总是矩形的，而不管图片内容如何。我们可以用具有自然形状的图片来创建应用程序和桌面程序。例如，对游戏软件的界面，我们希望有一个圆角按钮，那么可以用“有形窗口”来做到这一点。

一个有形窗口实际上就是 pixmap图像，它的背景颜色是透明的。这样，当背景图片有多种颜色时，我们就不会用不匹配的图标矩形边界来覆盖它。下面的例子在桌面上显示一个完整的独轮手推车。

```

/*示例开始—独轮手推车 wheelbarrow.c */

```

```

#include <gtk/gtk.h>

```

```

/* XPM */

```

```

static char * WheelbarrowFull_xpm[] = {

```

```

"48 48 64 1",

```

```

"      c None",

```

```

".      c #DF7DCF3CC71B",

```

```

"X      c #965875D669A6",

```

```

"o      c #71C671C671C6",

```

```

"O      c #A699A289A699",

```

```

"+      c #965892489658",

```

```

"@      c #8E38410330C2",

```



```
"# c #D75C7DF769A6",
"$ c #F7DECF3CC71B",
"% c #96588A288E38",
"& c #A69992489E79",
"* c #8E3886178E38",
"= c #104008200820",
"- c #596510401040",
"; c #C71B30C230C2",
": c #C71B9A699658",
"> c #618561856185",
", c #20811C712081",
"< c #104000000000",
"1 c #861720812081",
"2 c #DF7D4D344103",
"3 c #79E769A671C6",
"4 c #861782078617",
"5 c #41033CF34103",
"6 c #000000000000",
"7 c #49241C711040",
"8 c #492445144924",
"9 c #082008200820",
"0 c #69A618611861",
"q c #B6DA71C65144",
"w c #410330C238E3",
"e c #CF3CBAEAB6DA",
"r c #71C6451430C2",
"t c #EFBEDB6CD75C",
"y c #28A208200820",
"u c #186110401040",
"i c #596528A21861",
"p c #71C661855965",
"a c #A69996589658",
"s c #30C228A230C2",
"d c #BEFBA289AEBA",
"f c #596545145144",
"g c #30C230C230C2",
"h c #8E3882078617",
"j c #208118612081",
"k c #38E30C300820",
"l c #30C2208128A2",
"z c #38E328A238E3",
"x c #514438E34924",
"c c #618555555965",
"v c #30C2208130C2",
"b c #38E328A230C2",
"n c #28A228A228A2",
"m c #41032CB228A2",
"M c #104010401040",
"N c #492438E34103",
"B c #28A2208128A2",
"V c #A699596538E3",
"C c #30C21C711040",
```

```

"Z      c #30C218611040",
"A      c #965865955965",
"S      c #618534D32081",
"D      c #38E31C711040",
"F      c #082000000820",
"
"          .XoO
"      +@#$$%o&
"      *=-;#::o+
"          >,<12#:34
"          45671#:X3
"          +89<02qwo
"e*          >,<67;ro
"ty>          459@>+&&
"$2u+          ><ipas8*
"%$;=*          *3:..Xa.dfg>
"Oh$;ya          *3d.a8j,Xe.d3g8+
" Oh$;ka          *3d$a8lz,,xxc:..e3g54
" Oh$;ko          *pd$%svbzz,sxxxxxfX..&wn>
" Oh$@mO          *3dthwlsslslszjzxxxxxxxxx3:td8M4
" Oh$@g& *3d$XN1vvvlllm,mNwxxxxxxxxxfa.:,B*
" Oh$@,Od.cz1llllzlmmqV@V#V@fxxxxxxxxxf:%j5&
" Oh$1hd5lllsl1lCCZrV#r#:#2AxxxxxxxxxxcdwM*
" OXq6c.%8vvvlllZZiqqApA:mq:Xxcpcxxxxxxfdc9*
" 2r<6gde3blllZrVi7S@SV77A::qApxxxxxxfdcM
" : ,q-6MN.dfmZZrSS:#riirDSAX@Af5xxxxxxfevo",
" +A26jguXtAZZZC7iDiCCrVVii7Cmmmmxxxxxx%3g",
" *#16jszN..3DZZZZrCVSA2rZrV7Dmmwxxxx&en",
" p2yFvzssXe:fcZZCiId7iiZDiDSSZwxxx8e*>",
" OA1<jzxwxc:$d%NDZZZZCCZCCZCCZCmxxfd.B
" 3206Bwxxszx%et.eaAp77m77mmmf3&eeeg*
" @26MvzxNzvlbwfpdettttttttttt.c,n&
" *;16=lsNwNwgsvs1bwvccc3pcfu<o
" p;<69Bvwssszs1llbBl1ll1lllu<5+
" OS0y6FB1vvvzvzss,u=Bl1lj=54
" c1-699Blv1ll1lllu7k96MMmg4
" *10y8n6Fjv1ll1lB<166668
" S-kg+>666<M<996-y6n<8*
" p71=4 m69996kD8Z-66698&&
" &i0ycm6n4 ogk17,0<6666g
" N-k-<>          >=01-kuu666>
" ,6ky&          &46-10ul,66,
" Ou0<>          o66y<ulw<66&
"      *kk5          >66By7=xu664
"      <<M4          466lj<Mxu66o
"      *>>          +66uv,zN666*
"          566,xxj669
"          4666FF666>
"          >966666M
"          oM6668+
"          *4

```



```
GDK_BUTTON_PRESS_MASK );
gtk_signal_connect( GTK_OBJECT(window), "button_press_event",
                    GTK_SIGNAL_FUNC(close_application), NULL );
```

9.7 标尺构件GtkRuler

GtkRuler(标尺构件)一般用于在给定窗口中指示鼠标指针的位置。一个窗口可以有一个横跨整个窗口宽度的水平标尺和一个占据整个窗口高度的垂直标尺。标尺上有一个小三角形的指示器标出鼠标指针相对于标尺的精确位置。

有两种标尺构件：GtkHRuler（水平）和GtkVRuler（垂直）。

首先，必须创建标尺构件。水平和垂直标尺用下面的函数创建：

```
GtkWidget *gtk_hruler_new(void水平标尺 */
GtkWidget *gtk_vruler_new(void垂直标尺 */
```

一旦创建了标尺，我们就能指定它的度量单位。标尺的度量单位可以是 `GTK_PIXELS`，`GTK_INCHES`或 `GTK_CENTIMETERS`。可以用下面的函数设置：

```
void gtk_ruler_set_metric( GtkRuler *ruler,
                           GtkMetricType metric );
```

默认的度量单位是 `GTK_PIXELS`。

```
gtk_ruler_set_metric( GTK_RULER(ruler), GTK_PIXELS );
```

标尺构件的另一个重要属性是怎样标志刻度单位以及位置指示器一开始应该放在哪里。可以用下面的函数设置：

```
void gtk_ruler_set_range( GtkRuler *ruler,
                           gfloat lower,
                           gfloat upper,
                           gfloat position,
                           gfloat max_size );
```

其中`lower`和`upper`参数定义标尺的范围，`max_size`是要显示的最大可能数值。第四个参数`position`定义了标尺的指针指示器的初始位置。

垂直标尺能跨越800像素宽的窗口，因此：

```
gtk_ruler_set_range( GTK_RULER(vruler), 0, 800, 0, 800);
```

标尺上显示标志会从0到800，每100个像素一个数字。如果我们想让标尺的范围从7到16，可以使用下面的代码：

```
gtk_ruler_set_range( GTK_RULER(vruler), 7, 16, 0, 20);
```

标尺上的指示器是一个小三角形的标记，指示鼠标指针相对于标尺的位置。如果标尺是用于跟踪鼠标器指针的，应该将 `motion_notify_event`信号连接到标尺的 `motion_notify_event`方法。要跟踪鼠标在整个窗口区域内的移动，应该这样做：

```
#define EVENT_METHOD(i, x) GTK_WIDGET_CLASS(GTK_OBJECT(i)->klass)->x
gtk_signal_connect_object( GTK_OBJECT(area), "motion_notify_event",
                           (GtkSignalFunc)EVENT_METHOD(ruler, motion_notify_event),
                           GTK_OBJECT(ruler) );
```

下列例子创建一个绘图区，上面加一个水平标尺，左边加一个垂直标尺。绘图区的大小是 600像素宽×400像素高。水平标尺范围是从7到13，每100像素加一个刻度；垂直标尺范围从0到400，每100像素加一个刻度。

绘图区和标尺的定位是用表格构件 (GtkTable) 实现的。

```
/* 标尺示例开始 rulers.c */

#include <gtk/gtk.h>

#define EVENT_METHOD(i, x) GTK_WIDGET_CLASS(GTK_OBJECT(i)->klass)->x

#define XSIZE 600
#define YSIZE 400

/* 当点击"close"按钮时,退出应用程序*/
void close_application( GtkWidget *widget,
                        GdkEvent *event, gpointer data )
{
    gtk_main_quit();
}

/* 主函数 */
int main( int argc, char *argv[] ) {
    GtkWidget *window, *table, *area, *hrule, *vrule;

    /* 初始化GTK,创建主窗口*/
    gtk_init( &argc, &argv );

    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC( close_application ), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个GtkTable构件,标尺和绘图区放在里面*/
    table = gtk_table_new( 3, 2, FALSE );
    gtk_container_add( GTK_CONTAINER(window), table );

    area = gtk_drawing_area_new();
    gtk_drawing_area_size( (GtkDrawingArea *)area, XSIZE, YSIZE );
    gtk_table_attach( GTK_TABLE(table), area, 1, 2, 1, 2,
                      GTK_EXPAND|GTK_FILL, GTK_FILL, 0, 0 );
    gtk_widget_set_events( area, GDK_POINTER_MOTION_MASK |
                           GDK_POINTER_MOTION_HINT_MASK );

    /* 水平标尺放在顶部。鼠标移动穿过绘图区时,一个
     * motion_notify_event被传递给标尺的相应的事件处理函数*/
    hrule = gtk_hruler_new();
    gtk_ruler_set_metric( GTK_RULER(hrule), GTK_PIXELS );
    gtk_ruler_set_range( GTK_RULER(hrule), 7, 13, 0, 20 );
    gtk_signal_connect_object( GTK_OBJECT(area), "motion_notify_event",
                              (GtkSignalFunc)EVENT_METHOD(hrule,
                              motion_notify_event),
                              GTK_OBJECT(hrule) );

    /* GTK_WIDGET_CLASS(GTK_OBJECT(hrule)->klass)->motion_notify_event, */
}
```

```

gtk_table_attach( GTK_TABLE(table), hrule, 1, 2, 0, 1,
                  GTK_EXPAND|GTK_SHRINK|GTK_FILL, GTK_FILL, 0, 0 );
/* 垂直标尺显示在左边。当鼠标移动穿过绘图区时,
 * motion_notify_event被传递到标尺相应的事件处理函数中 */
vrule = gtk_vruler_new();
gtk_ruler_set_metric( GTK_RULER(vrule), GTK_PIXELS );
gtk_ruler_set_range( GTK_RULER(vrule), 0, YSIZE, 10, YSIZE );
gtk_signal_connect_object( GTK_OBJECT(area), "motion_notify_event",
                          (GtkSignalFunc)
                          GTK_WIDGET_CLASS(GTK_OBJECT(vrule)->klass)->
                          motion_notify_event,
                          GTK_OBJECT(vrule) );
gtk_table_attach( GTK_TABLE(table), vrule, 0, 1, 1, 2,
                  GTK_FILL, GTK_EXPAND|GTK_SHRINK|GTK_FILL, 0, 0 );

/* 现在显示所有的构件 */
gtk_widget_show( area );
gtk_widget_show( hrule );
gtk_widget_show( vrule );
gtk_widget_show( table );
gtk_widget_show( window );
gtk_main();

return(0);
} /* 示例结束 */

```

将上面的代码保存为 rulers.c，然后编写一个如下所示的 Makefile 文件。

```

CC = gcc
rulers: rulers.c
    $(CC) `gtk-config --cflags` rulers.c -o rulers \
        `gtk-config --libs`
clean:
    rm -f *.o rulers

```

编译，运行结果如图 9-4 所示。当鼠标在绘图区移动时，水平和垂直标尺上都能清楚地显示鼠标的位置。

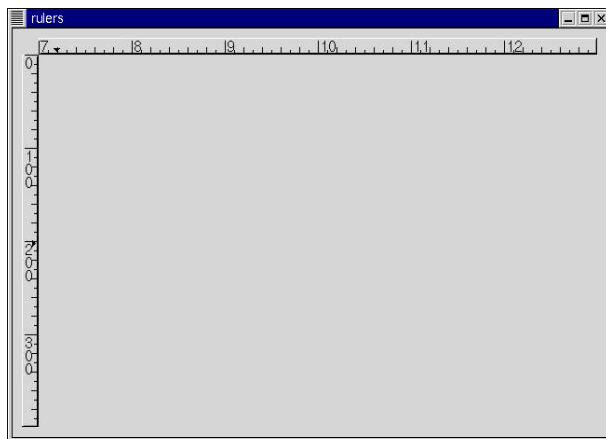


图9-4 标尺示例

9.8 文本输入构件GtkEntry

GtkEntry(文本输入构件)允许在一个单行文本框里输入和显示一行文本。文本可以用函数进行操作，如将新的文本替换、前插、追加到文本输入构件的当前内容中。

有两个用于创建文本输入构件的函数：

```
GtkWidget *gtk_entry_new(void);  
GtkWidget *gtk_entry_new_with_max_length( guint16 max);
```

第一个函数创建新的文本输入构件，第二个函数创建新的最大文本长度为 max的文本输入构件。

有几个函数是用来改变构件内的文本内容的：

```
void gtk_entry_set_text( GtkEntry *entry,  
                        const gchar *text );  
void gtk_entry_append_text( GtkEntry *entry,  
                           const gchar *text );  
void gtk_entry_prepend_text( GtkEntry *entry,  
                            const gchar *text
```

函数gtk_entry_set_text设置文本输入构件内的文本内容目录。

函数gtk_entry_append_text和gtk_entry_prepend_text将新文本前插和追加到构件当前文本中。

下面的函数用于设置当前插入点。

```
void gtk_entry_set_position( GtkEntry *entry,  
                           gint position );
```

文本输入构件内的文本可以用下面的函数获取。这在下面介绍的回调函数中是很有用的。

```
gchar *gtk_entry_get_text( GtkEntry *entry );
```

函数返回的值在函数内部使用，不必用 free()或者g_free()释放。如果想让文本输入构件是只读的，可以改变它的可编辑状态。

```
void gtk_entry_set_editable( GtkEntry *entry,  
                           gboolean editable );
```

其中editable可设为TRUE（可编辑）或FALSE（不可编辑）。

如果想让文本输入构件输入的文本不回显（比如用于接收口令），可以使用下列函数，其中visible可以取为TRUE（显示）或FALSE（不显示）。

```
void gtk_entry_set_visibility( GtkEntry *entry,  
                              gboolean visible );
```

文本内的某一部分可以用下面的函数设置为选中。一般用于用文本输入构件为用户提供一个缺省值，这样当构件获得焦点时就可以全部选中其中的文本，用户的输入会直接替换全部文本。

```
void gtk_entry_select_region( GtkEntry *entry,  
                             gint start,  
                             gint end );
```

如果我们想在用户输入文本时进行响应，可以为 activate或者changed信号设置回调函数。当用户在文本输入构件内部按回车键时引发 activate信号；当文本构件内部的文本发生变化时引发changed信号。

下面的代码是一个使用文本输入构件的示例。

```
/* 文本输入构件示例开始 entry.c */
#include <gtk/gtk.h>
void enter_callback(GtkWidget *widget, GtkWidget *entry)
{
    gchar *entry_text;
    entry_text = gtk_entry_get_text(GTK_ENTRY(entry));
    printf("Entry contents: %s\n", entry_text);
}

void entry_toggle_editable (GtkWidget *checkboxbutton,
                             GtkWidget *entry)
{
    gtk_entry_set_editable(GTK_ENTRY(entry),
                           GTK_TOGGLE_BUTTON(checkboxbutton)->active);
}

void entry_toggle_visibility (GtkWidget *checkboxbutton,
                              GtkWidget *entry)
{
    gtk_entry_set_visibility(GTK_ENTRY(entry),
                             GTK_TOGGLE_BUTTON(checkboxbutton)->active);
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *vbox, *hbox;
    GtkWidget *entry;
    GtkWidget *button;
    GtkWidget *check;
    gtk_init (&argc, &argv);

    /* 创建主窗口 */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_usize( GTK_WIDGET (window), 200, 100);
    gtk_window_set_title(GTK_WINDOW (window), "GTK Entry");
    gtk_signal_connect(GTK_OBJECT (window), "delete_event",
                      (GtkSignalFunc) gtk_exit, NULL);
    vbox = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    gtk_widget_show (vbox);
    entry = gtk_entry_new_with_max_length (50);
    gtk_signal_connect(GTK_OBJECT(entry), "activate",
                      GTK_SIGNAL_FUNC(enter_callback),
                      entry);
    gtk_entry_set_text (GTK_ENTRY (entry), "hello");
    gtk_entry_append_text (GTK_ENTRY (entry), " world");
    gtk_entry_select_region (GTK_ENTRY (entry),
                            0, GTK_ENTRY(entry)->text_length);
}
```



```

gtk_box_pack_start (GTK_BOX (vbox), entry, TRUE, TRUE, 0);
gtk_widget_show (entry);
hbox = gtk_hbox_new (FALSE, 0);
gtk_container_add (GTK_CONTAINER (vbox), hbox);
gtk_widget_show (hbox);

check = gtk_check_button_new_with_label("Editable");
gtk_box_pack_start (GTK_BOX (hbox), check, TRUE, TRUE, 0);
gtk_signal_connect (GTK_OBJECT(check), "toggled",
                    GTK_SIGNAL_FUNC(entry_toggle_editable),
                    entry);
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(check), TRUE);
gtk_widget_show (check);
check = gtk_check_button_new_with_label("Visible");
gtk_box_pack_start (GTK_BOX (hbox), check, TRUE, TRUE, 0);
gtk_signal_connect (GTK_OBJECT(check), "toggled",
                    GTK_SIGNAL_FUNC(entry_toggle_visibility),
                    entry);
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(check), TRUE);
gtk_widget_show (check);
button = gtk_button_new_with_label ("Close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC(gtk_exit),
                           GTK_OBJECT (window));

gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
gtk_widget_grab_default (button);
gtk_widget_show (button);
gtk_widget_show(window);
gtk_main();

return(0);
}

```

/*示例结束 */

将上面的代码保存为entry.c，然后编写一个如下所示的Makefile文件。

```

CC = gcc
entry: entry.c
    $(CC) `gtk-config --cflags` entry.c -o \
        entry `gtk-config --libs`
clean:
    rm -f *.o entry

```

编译应用程序，然后在 shell 提示符下输入 ./entry，执行结果如图9-5所示。当 Editable 检查按钮是按下状态时，可以编辑文本输入构件内的文本；如果该检查按钮是弹起的，不能对文本输入构件的文本进行编辑。Visible 检查按钮是按下状态时，用户输入的字符会显示在文本输入构件上，否则不显示出来。



图9-5 文本输入构件GtkEntry

9.9 微调按钮构件GtkSpinButton

GtkSpinButton(微调按钮构件)通常用于让用户从一个取值范围内选择一个值。它由一个文本输入框和旁边的向上和向下两个按钮组成。点击某一个按钮会让文本输入框内的数值大小在一定范围内改变。文本输入框也可以直接进行编辑。

微调按钮构件允许其中的数值没有小数位或具有指定的小数位，并且数值可以按一种可配置的方式增加或减小。在按钮较长时间呈按下状态时，构件的数值会根据工具按下时间的长短加速变化。

微调按钮用一个调整对象来维护该按钮能够取值的范围。微调按钮构件因此而具有了很强大的功能。

下面是创建调整对象的函数。这里的用意是展示其中所包含的数值的意义：

```

GtkWidget *gtk_adjustment_new( gfloat value,
                                gfloat lower,
                                gfloat upper,
                                gfloat step_increment,
                                gfloat page_increment,
                                gfloat page_size );

```

调整对象的这些属性在微调按钮构件中有如下用处：

value: 微调按钮构件的初值。

lower: 构件允许的最小值。

upper: 构件允许的最大值。

step_increment: 当鼠标左键按下时构件一次增加/减小的值。

page_increment: 当鼠标右键按下时构件一次增加/减小的值。

page size: 没有用到。

另外，当用鼠标中间键点击向下或向上按钮时，可以直接跳到构件的 lower或upper值。

用下面的函数创建新微调按钮构件：

```
GtkWidget *gtk_spin_button_new( GtkAdjustment *adjustment,
                                gfloat          climb_rate,
                                quint          digits );
```

其中的climb_rate参数是介于0.0和1.0间的值，指明构件数值变化的加速度（长时间按住按钮，数值会加速变化）。第三个参数digits指定要显示的值的小数位数。

创建微调按钮构件之后，还可以用下面的函数对其重新配置：

```
void gtk_spin_button_configure( GtkSpinButton *spin_button,
                               GtkAdjustment *adjustment,
                               gfloat         climb_rate,
                               quint          digits );
```

其中spin button参数就是要重新配置的参数。其他的参数与创建时的参数意思相同。

使用下面的函数可以设置或获取构件内部使用的调整对象：

[illegible]

```
GtkAdjustment adjustment(
    GtkSpinButton *spin_button );
```

显示数值的小数位数可以用下面的函数改变，其中 digits 就是小数位数：

```
void gtk_spin_button_set_digits( GtkSpinButton *spin_button,  
                                guint           digits ) ;
```

当前显示的构件数值可以用下面的函数改变：

```
void gtk_spin_button_set_value( GtkSpinButton *spin_button,  
                                gfloat         value ) ;
```

微调按钮构件的当前值可以以整数或浮点数的形式获得。使用下面的函数：

```
gfloat gtk_spin_button_get_value_as_float( GtkSpinButton *spin_button ) ;  
gint   gtk_spin_button_get_value_as_int(   GtkSpinButton *spin_button ) ;
```

如果想以当前值为基数改变构件的值，可以使用下面的函数：

```
void gtk_spin_button_spin( GtkSpinButton *spin_button,  
                           GtkSpinType   direction,  
                           gfloat        increment ) ;
```

其中，direction 参数可以取下面的值：

```
GTK_SPIN_STEP_FORWARD  
GTK_SPIN_STEP_BACKWARD  
GTK_SPIN_PAGE_FORWARD  
GTK_SPIN_PAGE_BACKWARD  
GTK_SPIN_HOME  
GTK_SPIN_END  
GTK_SPIN_USER_DEFINED
```

这个函数中包含的一些功能将在下面详细介绍。其中的许多设置都使用了与微调按钮构件相关联的调整对象的值。

GTK_SPIN_STEP_FORWARD 和 GTK_SPIN_STEP_BACKWARD 将构件的值按 increment 参数指定的数值增大或减小，除非 increment 参数是 0。这种情况下，构件的值将按与其相关联的调整对象的 step_increment 值改变。

GTK_SPIN_PAGE_FORWARD 和 GTK_SPIN_PAGE_BACKWARD 按 increment 参数改变微调按钮构件的值。

GTK_SPIN_HOME 将构件的值设置为相关联调整对象的范围的最小值。

GTK_SPIN_END 将构件的值设置为相关联调整对象的范围的最大值。

GTK_SPIN_USER_DEFINED 按指定的数值改变构件的数值。

下面介绍影响微调按钮构件的外观和行为的函数。

要介绍的第一个函数就是限制微调按钮构件的文本框只能输入数值。这样就阻止了用户输入任何非法的字符：

```
void gtk_spin_button_set_numeric( GtkSpinButton *spin_button,  
                                  gboolean       numeric ) ;
```

可以设置让微调按钮构件在 upper 和 lower 之间循环。也就是当达到最大值后再向上调整回到最小值，当达到最小值后再向下调整变为最大值。可以用下面的函数实现，其中 wrap 可以设置为 TRUE 和 FALSE：

```
void gtk_spin_button_set_wrap( GtkSpinButton *spin_button,  
                               gboolean       wrap ) ;
```

可以设置让微调按钮构件将其值圆整到最接近 `step_increment` 的值（在该微调按钮构件使用的调整对象中设置的）。用下面的函数实现：

```
void gtk_spin_button_set_snap_to_ticks( GtkSpinButton *spin_button,
                                         gboolean          snap_to_ticks );
```

微调按钮构件的更新方式可以用下面的函数改变：

```
void gtk_spin_button_set_update_policy( GtkSpinButton *spin_button,
                                         GtkSpinButtonUpdatePolicy policy );
```

其中 `policy` 参数可以取 `GTK_UPDATE_ALWAYS` 或 `GTK_UPDATE_IF_VALID`。

这些更新方式影响微调按钮构件在解析插入文本并将其值与调整对象的值同步时的行为。

在 `policy` 值为 `GTK_UPDATE_IF_VALID` 时，微调按钮构件只有在输入文本是其中调整对象指定范围内合法的值时才进行更新，否则文本会被重置为当前的值。

在 `policy` 值为 `GTK_UPDATE_ALWAYS` 时，在将文本转换为数值时忽略错误。

构件中使用的向上和向下的按钮的外观可以用下面的函数改变：

```
void gtk_spin_button_set_shadow_type( GtkSpinButton *spin_button,
                                       GtkShadowType  shadow_type );
```

与其他构件一样，`shadow_type` 参数可以取以下的值：

`GTK_SHADOW_IN`

`GTK_SHADOW_OUT`

`GTK_SHADOW_ETCHED_IN`

`GTK_SHADOW_ETCHED_OUT`

最后，可以强行要求微调按钮构件更新自己：

```
void gtk_spin_button_update( GtkSpinButton *spin_button );
```

下面是一个使用微调按钮构件的示例。

```
/* 微调按钮构件示例开始 spinbutton.c */
```

```
#include <gtk/gtk.h>
```

```
static GtkWidget *spinner1;
```

```
void toggle_snap( GtkWidget *widget,
                  GtkSpinButton *spin )
{
    gtk_spin_button_set_snap_to_ticks (spin,
                                         GTK_TOGGLE_BUTTON (widget)->active);
}
```

```
void toggle_numeric( GtkWidget *widget,
                    GtkSpinButton *spin )
{
    gtk_spin_button_set_numeric (spin,
                                   GTK_TOGGLE_BUTTON (widget)->active);
}
```

```
void change_digits( GtkWidget *widget,
                   GtkSpinButton *spin )
```

```
{
    gtk_spin_button_set_digits (GTK_SPIN_BUTTON (spinner1),
                                gtk_spin_button_get_value_as_int (spin));
}

void get_value( GtkWidget *widget,
                gpointer data )
{
    gchar buf[32];
    GtkLabel *label;
    GtkSpinButton *spin;

    spin = GTK_SPIN_BUTTON (spinner1);
    label = GTK_LABEL (gtk_object_get_user_data (GTK_OBJECT (widget)));
    if (GPOINTER_TO_INT (data) == 1)
        sprintf (buf, "%d", gtk_spin_button_get_value_as_int (spin));
    else
        sprintf (buf, "%.5f", spin->digits,
                  gtk_spin_button_get_value_as_float (spin));
    gtk_label_set_text (label, buf);
}

int main( int   argc,
          char *argv[] )
{
    GtkWidget *window;
    GtkWidget *frame;
    GtkWidget *hbox;
    GtkWidget *main_vbox;
    GtkWidget *vbox;
    GtkWidget *vbox2;
    GtkWidget *spinner2;
    GtkWidget *spinner;
    GtkWidget *button;
    GtkWidget *label;
    GtkWidget *val_label;
    GtkAdjustment *adj;

    /* 初始化GTK */
    gtk_init(&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (gtk_main_quit),
                        NULL);

    gtk_window_set_title (GTK_WINDOW (window), "Spin Button");

    main_vbox = gtk_vbox_new (FALSE, 5);
    gtk_container_set_border_width (GTK_CONTAINER (main_vbox), 10);
```

[illegible]

```
spinner = gtk_spin_button_new (adj, 0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner), FALSE);
gtk_spin_button_set_shadow_type (GTK_SPIN_BUTTON (spinner),
                                GTK_SHADOW_IN);
gtk_widget_set_usize (spinner, 55, 0);
gtk_box_pack_start (GTK_BOX (vbox2), spinner, FALSE, TRUE, 0);

frame = gtk_frame_new ("Accelerated");
gtk_box_pack_start (GTK_BOX (main_vbox), frame, TRUE, TRUE, 0);

vbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
gtk_container_add (GTK_CONTAINER (frame), vbox);

hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Value :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (0.0, -10000.0, 10000.0,
                                           0.5, 100.0, 0.0);

spinner1 = gtk_spin_button_new (adj, 1.0, 2);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner1), TRUE);
gtk_widget_set_usize (spinner1, 100, 0);
gtk_box_pack_start (GTK_BOX (vbox2), spinner1, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Digits :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (2, 1, 5, 1, 1, 0);
spinner2 = gtk_spin_button_new (adj, 0.0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner2), TRUE);
gtk_signal_connect (GTK_OBJECT (adj), "value_changed",
                   GTK_SIGNAL_FUNC (change_digits),
                   (gpointer) spinner2);
gtk_box_pack_start (GTK_BOX (vbox2), spinner2, FALSE, TRUE, 0);

hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);

button = gtk_check_button_new_with_label ("Snap to 0.5-ticks");
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (toggle_snap),
```

```

        spinner1);
gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);

button = gtk_check_button_new_with_label ("Numeric only input mode");
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (toggle_numeric),
                    spinner1);
gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);

val_label = gtk_label_new ("");

hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);
button = gtk_button_new_with_label ("Value as Int");
gtk_object_set_user_data (GTK_OBJECT (button), val_label);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (get_value),
                    GINT_TO_POINTER (1));
gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

button = gtk_button_new_with_label ("Value as Float");
gtk_object_set_user_data (GTK_OBJECT (button), val_label);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (get_value),
                    GINT_TO_POINTER (2));
gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

gtk_box_pack_start (GTK_BOX (vbox), val_label, TRUE, TRUE, 0);
gtk_label_set_text (GTK_LABEL (val_label), "0");
hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (main_vbox), hbox, FALSE, TRUE, 0);
button = gtk_button_new_with_label ("Close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),
                           GTK_OBJECT (window));
gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

gtk_widget_show_all (window); /* Enter the e
*/ gtk_main ();
return(0);}

/*示例结束 */

```

图9-6是上面代码的运行结果。点击微调按钮右边的向上和向下箭头，前面的文本框的数值会随之而改变。

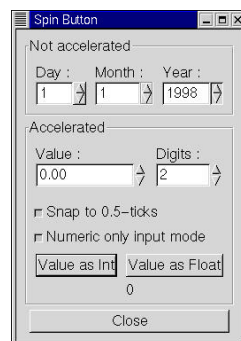


图9-6 微调按钮构件示例

9.10 组合框GtkCombo

GtkCombo(组合框)是极为常见的构件，实际上它仅仅是其他构件的集合。从用户的观点来说，这个构件是由一个文本输入构件和一个下拉菜单组成的，用户可以从一个预先定义的

列表里面选择一个选项，同时，用户也可以直接在文本框里面输入文本。

下面是从定义组合框构件的结构里面摘取出来的，从中可以看到组合框构件是由什么构件组合形成的：

```
struct _GtkCombo
{
    GtkHBox hbox;
    GtkWidget *entry;
    GtkWidget *button;
    GtkWidget *popup;
    GtkWidget *popwin;
    GtkWidget *list;
    ... };
```

可以看到，组合框构件有两个主要部分：一个输入框和一个列表。

用下面的函数创建组合框构件：

```
GtkWidget *gtk_combo_new( void );
```

现在，如果想设置显示在输入框部分中的字符串，可以直接操纵组合框构件内部的文本输入构件：

```
gtk_entry_set_text(GTK_ENTRY(GTK_COMBO(combo)->entry), "My String.");
```

要设置下拉列表中的值，可以使用下面的函数：

```
void gtk_combo_set_popdown_strings( GtkCombo *combo,
                                     GList      *strings );
```

在使用这个函数之前，先得将要添加的字符串组合成一个 GList链表。GList是一个双向链表，是Glib的一部分。要做的就是设置一个 GList指针，其值设为NULL，然后用下面的函数将字符串追加到链表当中：

```
GList *g_list_append( GList *glist,
                      gpointer data );
```

要注意的是：一定要将 GList链表的初值设为NULL，必须将g_list_append函数返回的值赋给要操作的链表本身。

下面是一段典型的代码，用于创建一个选项列表：

```
GList *glist=NULL;

glist = g_list_append(glist, "String 1");
glist = g_list_append(glist, "String 2");
glist = g_list_append(glist, "String 3");
glist = g_list_append(glist, "String 4");

gtk_combo_set_popdown_strings( GTK_COMBO(combo), glist );
```

到这里为止，你现在已经可以使用设置的组合框构件了。有几个行为是可以改变的。下面是相关的函数：

```
void gtk_combo_set_use_arrows( GtkCombo *combo,
                               gint       val );

void gtk_combo_set_use_arrows_always( GtkCombo *combo,
                                       gint       val );

void gtk_combo_set_case_sensitive( GtkCombo *combo,
                                   gint       val );
```

`gtk_combo_set_use_arrows()`让用户用上/下方向键改变文本输入构件内的值。这并没有改变列表的值，只是用列表中的下一个列表项替换了文本输入框中的文本（向上则取上一个值，向下则取下一个值）。这是通过搜索当前项在列表中的位置并选择前一项 /下一项来实现的。通常，在一个输入框中方向键是用来改变焦点的（也可以用 `TAB`键）。注意，如果当前项是列表的最后一项，按向下的方向键会改变焦点的位置（这对列表在第一项时按向上方向键也适用）。

如果当前值并不在列表中，则不能使用 `gtk_combo_set_use_arrows()`函数。

同样地，`gtk_combo_set_use_arrows_always()`允许使用上/下方向键在下拉列表中选择列表项，但是它在列表项中循环，也就是当列表项位于第一个表项时按向上方向键，会跳到最后一个，当列表项位于最后一个表项时按向下方向键，会跳到第一个。这样可以完全禁止使用方向键改变焦点。

`gtk_combo_set_case_sensitive()`函数切换GTK是否以大小写敏感的方式搜索其中的列表项。这一般用在内部文本输入构件中的文本查找组合框构件中的列表值。可以将其设置为大小写敏感或不敏感。如果用户同时按下“`Alt`”和“`Tab`”键，组合框构件还可以用来完成当前输入。注意，窗口管理器也要使用这种组合键方式，将会忽略GTK中这个组合键的使用。

注意，我们使用的是组合框构件，它能够为我们从一个下拉列表中选择一项。这一点是很直截了当的。大多数时候，你可能很关心怎样从其中的文本输入构件中获取数据。组合框构件内部的文本输入构件可以用 `GTK_ENTRY(GTK_COMBO(combo)->entry)`访问。一般想要做的两件主要工作一个是连接一个 `activate`，当用户按回车键时能够进行响应，另一个就是读出其中的文本。第一件工作可以用下面的方法实现：

```
gtk_signal_connect(GTK_OBJECT(GTK_COMBO(combo)->entry), "activate",
                  GTK_SIGNAL_FUNC (my_callback_function), my_data);
```

可以使用下面的函数在任意时候取得文本输入构件中的文本：

```
gchar *gtk_entry_get_text(GtkEntry *entry);
```

具体做法如下：

```
char *string;
string = gtk_entry_get_text(GTK_ENTRY(GTK_COMBO(combo)->entry));
```

这就是取得文本输入框中字符串的方法。

```
void gtk_combo_disable_activate(GtkCombo *combo);
```

这个函数禁用组合框构件内部的文本输入构件（`GtkEntry`）的`activate`信号。

9.11 日历构件GtkCalendar

`GtkCalendar`(日历构件)显示一个月历视图，可以在上面方便地选择年份、月份和日期。这样，如果要做与日期相关的编程，不再需要考虑复杂的历法问题。日历构件本身外观也很漂亮，创建和使用都非常简单。同时，日历构件 `GtkCalendar`不存在2000年问题。如果要开发一个日程管理等类的软件，这个构件是一个很好的选择。

创建日历构件的方法和其他构件的类似：

```
GtkWidget *gtk_calendar_new();
```

有时候，需要同时对构件的外观和内容做很多的修改。这时候可能会引起构件的多次更新，导致屏幕闪烁。可以在修改之前使用一个函数将构件“冻结”，然后在修改完成之后再

一个函数将构件“解冻”。这样，构件在整个过程中只做一次更新。

这个函数将构件“冻结”：

```
void gtk_calendar_freeze( GtkCalendar *Calendar );
```

这个函数将构件“解冻”：

```
void gtk_calendar_thaw ( GtkCalendar *Calendar );
```

这两个函数和其他构件（比如 GtkText）的冻结/解冻函数作用完全一样。

日历构件有几个选项，可以用来改变构件的外观和操作方式。使用下面的函数可以改变这些选项：

```
void gtk_calendar_display_options( GtkCalendar *calendar,  
                                   GtkCalendarDisplayOptions flags );
```

函数中的 flags 参数可以将下面的五种选项中的一个或者多个用逻辑位或（|）操作符组合起来：

GTK_CALENDAR_SHOW_HEADING：这个选项指定在绘制日历构件时，应该显示月份和年份。

GTK_CALENDAR_SHOW_DAY_NAMES：这个选项指定用三个字母的缩写显示每一天是星期几（比如 MON、TUE 等）。

GTK_CALENDAR_NO_MONTH_CHANGE：这个选项指定用户不应该也不能够改变显示的月份。如果只想显示某个特定的月份，则可以使用这个选项。比如，如果在窗口上同时为一年的12个月分别设置一个日历构件时。

GTK_CALENDAR_SHOW_WEEK_NUMBERS：这个选项指定应该在构件的左边显示每一周在全年的周序号（一年是52个周，元月1日是第1周，12月31日是第52周）。

GTK_CALENDAR_WEEK_START_MONDAY：这个选项指定在日历构件中每一周是从星期一开始而不是从星期天开始。缺省设置是从星期天开始。此选项只影响日期在构件中从左到右的排列顺序。

下面的函数用于设置当前要显示的日期：

```
gint gtk_calendar_select_month( GtkCalendar *calendar,  
                                guint         month,  
                                guint         year );
```

```
void gtk_calendar_select_day( GtkCalendar *calendar,  
                              guint         day );
```

gtk_calendar_select_month() 的返回值是一个布尔值，指示设置是否成功。如果设置一个非法值，比如31月，则会返回一个 FALSE 值。

使用 gtk_calendar_select_day() 函数，如果 day 参数指定的日期是合法的，会在日历构件中选中该日期。如果 day 参数的值是0，将清除当前的选择。

除了可以选中一个日期以外，在一个月中可以有任意个日期被“标记”。被“标记”的日期会在日历构件中高亮显示。下面的函数用于标记日期和取消标记：

```
gint gtk_calendar_mark_day( GtkCalendar *calendar,  
                            guint         day );
```

```
gint gtk_calendar_unmark_day( GtkCalendar *calendar,  
                              guint         day );
```

```
void gtk_calendar_clear_marks( GtkCalendar *calendar);
```

当前标记的日期存储在一个 GtkCalendar结构的数组中。数组的长度是 31，这样，要想知道某个特定的日期是否被标记，可以访问数值中相应的元素（注意，在C语言中，数值是从0开始编号的）。例如：

```
GtkCalendar *calendar;
calendar = gtk_calendar_new();
...
/* 当月7日被标记了吗？*/
if (calendar->marked_date[7-1])
    /* 若执行此处的代码，表明7日已经被标记 */
```

注意，在月份和年份变化时，被标记的日期是不会变化的。

下面的函数用于取得当前选中的年/月/日值：

```
void gtk_calendar_get_date( GtkCalendar *calendar,
                           guint          *year,
                           guint          *month,
                           guint          *day );
```

使用这个函数时，需要先声明几个 guint类型的变量——传递给函数的year、month和day参数。所需要的返回值就存放在这几个变量中。如果将某一个参数设置为 NULL，则不返回该值。

日历构件能够引发许多信号，用于指示日期被选中以及选择发生的变化。信号的意义很容易理解。信号名称如下：

```
month_changed          选择月份变化 */
day_selected           选择日期变化 */
day_selected_double_click 选中日期并以鼠标双击 */
prev_month             选择前一月 */
next_month             选择下一月 */
prev_year              选择前一年 */
next_year              选择下一年 */
```

下面是一个日历构件的示例，运用了上面介绍的各项特性。

```
/* 日历构件示例开始 calendar.c */
#include <gtk/gtk.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

#define DEF_PAD 10
#define DEF_PAD_SMALL 5

#define TM_YEAR_BASE 1900

typedef struct _CalendarData {
    GtkWidget *flag_checkboxes[5];
    gboolean settings[5];
    gchar      *font;
    GtkWidget *font_dialog;
    GtkWidget *window;
```

```

    GtkWidget *prev2_sig;
    GtkWidget *prev_sig;
    GtkWidget *last_sig;
    GtkWidget *month;
} CalendarData;

enum {
    calendar_show_header,
    calendar_show_days,
    calendar_month_change,
    calendar_show_week,
    calendar_monday_first
};

/*
 * GtkCalendar 日历构件
 */

void calendar_date_to_string( CalendarData *data,
                             char          *buffer,
                             gint          buff_len )
{
    struct tm tm;
    time_t time;

    memset (&tm, 0, sizeof (tm));
    gtk_calendar_get_date (GTK_CALENDAR(data->window),
                          &tm.tm_year, &tm.tm_mon, &tm.tm_mday);
    tm.tm_year -= TM_YEAR_BASE;
    time = mktime(&tm);
    strftime (buffer, buff_len-1, "%x", gmtime(&time));
}

void calendar_set_signal_strings( char          *sig_str,
                                 CalendarData *data )
{
    gchar *prev_sig;

    gtk_label_get (GTK_LABEL (data->prev_sig), &prev_sig);
    gtk_label_set (GTK_LABEL (data->prev2_sig), prev_sig);

    gtk_label_get (GTK_LABEL (data->last_sig), &prev_sig);
    gtk_label_set (GTK_LABEL (data->prev_sig), prev_sig);
    gtk_label_set (GTK_LABEL (data->last_sig), sig_str);
}

void calendar_month_changed( GtkWidget      *widget,
                             CalendarData *data )
{
    char buffer[256] = "month_changed: ";

```

```
calendar_date_to_string (data, buffer+15, 256-15);
calendar_set_signal_strings (buffer, data);
}

void calendar_day_selected( GtkWidget      *widget,
                           CalendarData *data )
{
    char buffer[256] = "day_selected: ";

    calendar_date_to_string (data, buffer+14, 256-14);
    calendar_set_signal_strings (buffer, data);
}

void calendar_day_selected_double_click( GtkWidget      *widget,
                                         CalendarData *data )
{
    struct tm tm;
    char buffer[256] = "day_selected_double_click: ";

    calendar_date_to_string (data, buffer+27, 256-27);
    calendar_set_signal_strings (buffer, data);

    memset (&tm, 0, sizeof (tm));
    gtk_calendar_get_date (GTK_CALENDAR(data->window),
                          &tm.tm_year, &tm.tm_mon, &tm.tm_mday);
    tm.tm_year -= TM_YEAR_BASE;

    if(GTK_CALENDAR(data->window)->marked_date[tm.tm_mday-1] == 0) {
        gtk_calendar_mark_day(GTK_CALENDAR(data->window),tm.tm_mday);
    } else
        gtk_calendar_unmark_day(GTK_CALENDAR(data->window),tm.tm_mday);
    }
}

void calendar_prev_month( GtkWidget      *widget,
                          CalendarData *data )
{
    char buffer[256] = "prev_month: ";

    calendar_date_to_string (data, buffer+12, 256-12);
    calendar_set_signal_strings (buffer, data);
}

void calendar_next_month( GtkWidget      *widget,
                          CalendarData *data )
{
    char buffer[256] = "next_month: ";

    calendar_date_to_string (data, buffer+12, 256-12);
    calendar_set_signal_strings (buffer, data);
}
```

```
void calendar_prev_year( GtkWidget      *widget,
                        CalendarData *data )
{
    char buffer[256] = "prev_year: ";

    calendar_date_to_string (data, buffer+11, 256-11);
    calendar_set_signal_strings (buffer, data);
}

void calendar_next_year( GtkWidget      *widget,
                        CalendarData *data )
{
    char buffer[256] = "next_year: ";

    calendar_date_to_string (data, buffer+11, 256-11);
    calendar_set_signal_strings (buffer, data);
}

void calendar_set_flags( CalendarData *calendar )
{
    gint i;
    gint options=0;
    for (i=0;i<5;i++)
        if (calendar->settings[i])
        {
            options=options + (1<<i);
        }
    if (calendar->window)
        gtk_calendar_display_options (GTK_CALENDAR (calendar->window), options);
}

void calendar_toggle_flag( GtkWidget      *toggle,
                        CalendarData *calendar )
{
    gint i;
    gint j;
    j=0;
    for (i=0; i<5; i++)
        if (calendar->flag_checkboxes[i] == toggle)
            j = i;

    calendar->settings[j]=!calendar->settings[j];
    calendar_set_flags(calendar);
}

void calendar_font_selection_ok( GtkWidget      *button,
                        CalendarData *calendar )
{
    GtkStyle *style;
    GdkFont  *font;
```

```

calendar->font = gtk_font_selection_dialog_get_font_name(
    GTK_FONT_SELECTION_DIALOG (calendar->font_dialog));
if (calendar->>window)
{
font=gtk_font_selection_dialog_get_font(GTK_FONT_SELECTION_DIALOG(calendar-
>font_dialog));
    if (font)
    {
        style = gtk_style_copy (gtk_widget_get_style (calendar->>window));
        gdk_font_unref (style->font);
        style->font = font;
        gdk_font_ref (style->font);
        gtk_widget_set_style (calendar->>window, style);
    }
}

void calendar_select_font( GtkWidget      *button,
                           CalendarData *calendar )
{
    GtkWidget *window;

    if (!calendar->font_dialog) {
        window = gtk_font_selection_dialog_new ("Font Selection Dialog");
        g_return_if_fail(GTK_IS_FONT_SELECTION_DIALOG(window));
        calendar->font_dialog = window;

        gtk_window_position (GTK_WINDOW (window), GTK_WIN_POS_MOUSE);

        gtk_signal_connect (GTK_OBJECT (window), "destroy",
            GTK_SIGNAL_FUNC (gtk_widget_destroyed),
            &calendar->font_dialog);

        gtk_signal_connect (GTK_OBJECT (GTK_FONT_SELECTION_DIALOG (window)-
>ok_button),
            "clicked", GTK_SIGNAL_FUNC(calendar_font_selection_ok),
            calendar);
        gtk_signal_connect_object (GTK_OBJECT (GTK_FONT_SELECTION_DIALOG (window)-
>cancel_button),
            "clicked",
            GTK_SIGNAL_FUNC (gtk_widget_destroy),
            GTK_OBJECT (calendar->font_dialog));
    }
    window=calendar->font_dialog;
    if (!GTK_WIDGET_VISIBLE (window))
        gtk_widget_show (window);
    else
        gtk_widget_destroy (window);
}

void create_calendar()

```



```
{
    GtkWidget *window;
    GtkWidget *vbox, *vbox2, *vbox3;
    GtkWidget *hbox;
    GtkWidget *hbbox;
    GtkWidget *calendar;
    GtkWidget *toggle;
    GtkWidget *button;
    GtkWidget *frame;
    GtkWidget *separator;
    GtkWidget *label;
    GtkWidget *bbox;
    static CalendarData calendar_data;
    gint i;

    struct {
        char *label;
    } flags[] =
    {
        { "Show Heading" },
        { "Show Day Names" },
        { "No Month Change" },
        { "Show Week Numbers" },
        { "Week Start Monday" }
    };

    calendar_data.window = NULL;
    calendar_data.font = NULL;
    calendar_data.font_dialog = NULL;

    for (i=0; i<5; i++) {
        calendar_data.settings[i]=0;
    }

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkCalendar Example");
    gtk_container_border_width (GTK_CONTAINER (window), 5);
    gtk_signal_connect(GTK_OBJECT(window), "destroy",
        GTK_SIGNAL_FUNC(gtk_main_quit),
        NULL);
    gtk_signal_connect(GTK_OBJECT(window), "delete-event",
        GTK_SIGNAL_FUNC(gtk_false),
        NULL);

    gtk_window_set_policy(GTK_WINDOW(window), FALSE, FALSE, TRUE);

    vbox = gtk_vbox_new(FALSE, DEF_PAD);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    /*
```

* 顶级窗口，其中包含日历构件，设置 flags选项的按钮和设置字体的按钮

*/

```
hbox = gtk_hbox_new(FALSE, DEF_PAD);
gtk_box_pack_start (GTK_BOX(vbox), hbox, TRUE, TRUE, DEF_PAD);
hbbox = gtk_hbutton_box_new();
gtk_box_pack_start(GTK_BOX(hbox), hbbox, FALSE, FALSE, DEF_PAD);
gtk_button_box_set_layout(GTK_BUTTON_BOX(hbbox), (GTK_BUTTONBOX_SPREAD);
gtk_button_box_set_spacing(GTK_BUTTON_BOX(hbbox), 5);

/* 日历构件 */
frame = gtk_frame_new("Calendar");
gtk_box_pack_start(GTK_BOX(hbbox), frame, FALSE, TRUE, DEF_PAD);
calendar=gtk_calendar_new();
calendar_data.window = calendar;
calendar_set_flags(&calendar_data);
gtk_calendar_mark_day ( GTK_CALENDAR(calendar), 19);
gtk_container_add( GTK_CONTAINER( frame), calendar);
gtk_signal_connect (GTK_OBJECT (calendar), "month_changed",
                    GTK_SIGNAL_FUNC (calendar_month_changed),
                    &calendar_data);
gtk_signal_connect (GTK_OBJECT (calendar), "day_selected",
                    GTK_SIGNAL_FUNC (calendar_day_selected),
                    &calendar_data);
gtk_signal_connect (GTK_OBJECT (calendar), "day_selected_double_click",
                    GTK_SIGNAL_FUNC (calendar_day_selected_double_click),
                    &calendar_data);
gtk_signal_connect (GTK_OBJECT (calendar), "prev_month",
                    GTK_SIGNAL_FUNC (calendar_prev_month),
                    &calendar_data);
gtk_signal_connect (GTK_OBJECT (calendar), "next_month",
                    GTK_SIGNAL_FUNC (calendar_next_month),
                    &calendar_data);
gtk_signal_connect (GTK_OBJECT (calendar), "prev_year",
                    GTK_SIGNAL_FUNC (calendar_prev_year),
                    &calendar_data);
gtk_signal_connect (GTK_OBJECT (calendar), "next_year",
                    GTK_SIGNAL_FUNC (calendar_next_year),
                    &calendar_data);

separator = gtk_vseparator_new ();
gtk_box_pack_start (GTK_BOX (hbox), separator, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new(FALSE, DEF_PAD);
gtk_box_pack_start(GTK_BOX(hbox), vbox2, FALSE, FALSE, DEF_PAD);

frame = gtk_frame_new("Flags");
gtk_box_pack_start(GTK_BOX(vbox2), frame, TRUE, TRUE, DEF_PAD);
vbox3 = gtk_vbox_new(TRUE, DEF_PAD_SMALL);
gtk_container_add(GTK_CONTAINER(frame), vbox3);
```

```

for (i = 0; i < 5; i++)
{
    toggle = gtk_check_button_new_with_label(flags[i].label);
    gtk_signal_connect (GTK_OBJECT (toggle),
                        "toggled",
                        GTK_SIGNAL_FUNC(calendar_toggle_flag),
                        &calendar_data);
    gtk_box_pack_start (GTK_BOX (vbox3), toggle, TRUE, TRUE, 0);
    calendar_data.flag_checkboxes[i]=toggle;
}
/* 创建一个按钮，用于设置字体 */
button = gtk_button_new_with_label("Font...");
gtk_signal_connect (GTK_OBJECT (button),
                    "clicked",
                    GTK_SIGNAL_FUNC(calendar_select_font),
                    &calendar_data);
gtk_box_pack_start (GTK_BOX (vbox2), button, FALSE, FALSE, 0);

frame = gtk_frame_new("Signal events");
gtk_box_pack_start(GTK_BOX(vbox), frame, TRUE, TRUE, DEF_PAD);

vbox2 = gtk_vbox_new(TRUE, DEF_PAD_SMALL);
gtk_container_add(GTK_CONTAINER(frame), vbox2);

hbox = gtk_hbox_new (FALSE, 3);
gtk_box_pack_start (GTK_BOX (vbox2), hbox, FALSE, TRUE, 0);
label = gtk_label_new ("Signal:");
gtk_box_pack_start (GTK_BOX (hbox), label, FALSE, TRUE, 0);
calendar_data.last_sig = gtk_label_new ("");
gtk_box_pack_start (GTK_BOX (hbox), calendar_data.last_sig, FALSE, TRUE, 0);

hbox = gtk_hbox_new (FALSE, 3);
gtk_box_pack_start (GTK_BOX (vbox2), hbox, FALSE, TRUE, 0);
label = gtk_label_new ("Previous signal:");
gtk_box_pack_start (GTK_BOX (hbox), label, FALSE, TRUE, 0);
calendar_data.prev_sig = gtk_label_new ("");
gtk_box_pack_start (GTK_BOX (hbox), calendar_data.prev_sig, FALSE, TRUE, 0);

hbox = gtk_hbox_new (FALSE, 3);
gtk_box_pack_start (GTK_BOX (vbox2), hbox, FALSE, TRUE, 0);
label = gtk_label_new ("Second previous signal:");
gtk_box_pack_start (GTK_BOX (hbox), label, FALSE, TRUE, 0);
calendar_data.prev2_sig = gtk_label_new ("");
gtk_box_pack_start (GTK_BOX (hbox), calendar_data.prev2_sig, FALSE, TRUE,

0);

bbox = gtk_hbutton_box_new ();
gtk_box_pack_start (GTK_BOX (vbox), bbox, FALSE, FALSE, 0);
gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox), GTK_BUTTONBOX_END);
button = gtk_button_new_with_label ("Close");

```

```

gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (gtk_main_quit),
                    NULL);
gtk_container_add (GTK_CONTAINER (bbox), button);
GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
gtk_widget_grab_default (button);

gtk_widget_show_all(window);
}

int main(int argc,
        char *argv[])
{
    gtk_set_locale ();
    gtk_init (&argc, &argv);

    create_calendar();

    gtk_main();

    return(0);
}
/* 示例结束 */

```

示例的运行效果如图9-7所示。其中的Flags框架中的几个检查按钮用于设置日历构件的属性。尝试点击这几个检查按钮，看看有什么效果。Font...按钮用于设置日历构件的字体。

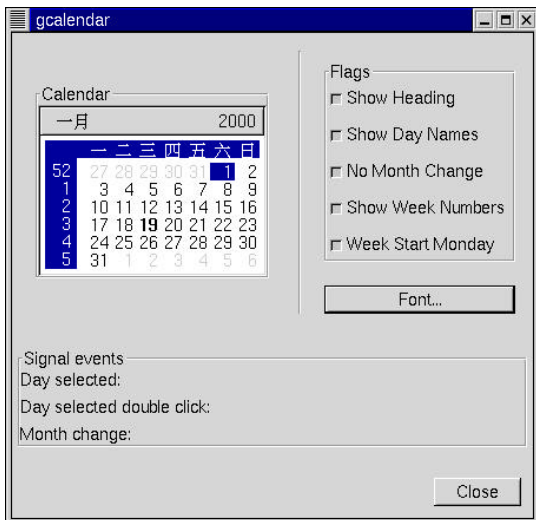


图9-7 日历构件GtkCalendar

9.12 颜色选择构件GtkColorSelect

GtkColorSelect(颜色选择构件)是一个用来交互式地选择颜色的构件。这个组合构件让用户通过操纵RGB值(红绿蓝)和HSV值(色调、饱和度、数值)来选择颜色。这是通过调整

GtkSlider构件的值或者文本输入构件的值，或者从一个色调 /饱和度 /数值条上选择相应的颜色来实现的。你还可以通过它来设置颜色的透明性。

目前，颜色选择构件只能引发一种信号：color_changed。它是在构件内的颜色值发生变化时，或者当用户通过 gtk_color_selection_set_color()函数显式设置构件的颜色值时引发。

现在可以看一下颜色选择构能够为我们提供一些什么。这个构件有两种风格：

gtk_color_selection和gtk_color_selection_dialog。

```
GtkWidget *gtk_color_selection_new( void );
```

这个函数很少用到。它创建一个孤立的颜色选择构件，并需要将其放在某个窗口上。颜色选择构件是从GtkVBox构件派生的。

```
GtkWidget *gtk_color_selection_dialog_new( const gchar *title );
```

这是最常用的颜色选择构件的构造函数，它创建一个颜色选择对话框。它内部有一个框架构件，框架构件中包含了一个颜色选择构件、一个垂直分隔线构件、一个 GtkHBox构件以及Ok、Cancel、Help三个按钮。你可以通过访问颜色选择对话框构件结构中的 ok_button、cancel_button和help_button构件来访问它们。例如：

```
GTK_COLOR_SELECTION_DIALOG(colorsel_dialog)->ok_button)
```

```
void gtk_color_selection_set_update_policy( GtkColorSelection *coloresel,  
                                             GtkUpdateType      policy );
```

这个函数设置颜色选择构件的更新方式。缺省的更新方式是GTK_UPDATE_CONTINUOUS，这意味着当用户拖动对话框内部的滑块，或者按下鼠标键并在色调 /饱和度轮形图或颜色值条上拖动时，当前颜色值将连续更新。如果碰到性能问题，可以将它设置为 GTK_UPDATE_DISCONTINUOUS或GTK_UPDATE_DELAYED。

```
void gtk_color_selection_set_opacity( GtkColorSelection *coloresel,  
                                      gint                use_opacity );
```

颜色选择构件支持调整颜色的不透明性（一般也称为alpha通道）。缺省值是禁用这个特性。调用下面的函数，将use_opacity设置为TRUE启用该特性。同样，use_opacity设置为FALSE时将禁用此特性。

```
void gtk_color_selection_set_color( GtkColorSelection *coloresel,  
                                    gdouble            *color );
```

可以调用这个函数显式地设置颜色选择构件的当前颜色。其中的 color参数是一个指向颜色值（gdouble类型）的数组。数组的长度依赖于是否启用了不透明性。数组的第一位包含红色值，1是绿色，2是蓝色，3是不透明性（只有不透明性启用时才有，创建前面的 gtk_color_selection_set_opacity()函数）。所有的值都介于0.0和1.0之间。

```
void gtk_color_selection_get_color( GtkColorSelection *coloresel,  
                                    gdouble            *color );
```

当需要查询当前颜色值时，典型情况是接收到一个 color_changed信号时，使用这个函数。其中，color值是一个指向颜色数组的指针。

下面是一个简单的示例，它演示了如何使用颜色选择对话框构件。这个程序显示了一个包含绘图区的窗口。点击它会打开一个颜色选择对话框，改变颜色选择对话框中的颜色，会

改变绘图区的背景色。

```
/* 颜色选择对话框示例开始colorsel.c */

#include <glib.h>
#include <gdk/gdk.h>
#include <gtk/gtk.h>

GtkWidget *colorseldlg = NULL;
GtkWidget *drawingarea = NULL;

/* 颜色改变信号的处理函数 */
void color_changed_cb (GtkWidget *widget, GtkColorSelection *colorsel)
{
    gdouble color[3];
    GdkColor gdk_color;
    GdkColormap *colormap;

    /* 获得绘图区的色彩表 */
    colormap = gdk_window_get_colormap (drawingarea->window);

    /* 获得当前颜色 */
    gtk_color_selection_get_color (colorsel, color);

    /* 将颜色值转换为16位无符号整数(0..65535)并
     * 插入到GdkColor结构中 */
    gdk_color.red = (guint16)(color[0]*65535.0);
    gdk_color.green = (guint16)(color[1]*65535.0);
    gdk_color.blue = (guint16)(color[2]*65535.0);

    /* 分配颜色 */
    gdk_color_alloc (colormap, &gdk_color);

    /* 设置窗口的背景颜色 */
    gdk_window_set_background (drawingarea->window, &gdk_color);

    /* 清除窗口 */
    gdk_window_clear (drawingarea->window);
}

/* 绘图区事件处理函数 */

gint area_event (GtkWidget *widget, GdkEvent *event, gpointer client_data)
{
    gint handled = FALSE;
    GtkWidget *colorsel;

    /*检查是否接收到一个按键按下事件 */

    if (event->type == GDK_BUTTON_PRESS && colorseldlg == NULL)
    {

```

```
/* 接收到了一个事件，但是还没有创建颜色选择对话框 */
handled = TRUE;

/* 创建颜色选择对话框 */
colourseldlg = gtk_color_selection_dialog_new("Select background color");

/* 获取颜色选择构件 */
colorsel = GTK_COLOR_SELECTION_DIALOG(colourseldlg)->colorsel;

/* 为"color_changed"信号设置回调函数，将用户数据设置为
 * 颜色选择构件 */
gtk_signal_connect(GTK_OBJECT(colorsel), "color_changed",
    (GtkSignalFunc)color_changed_cb, (gpointer)colorsel);

/* 显示对话框 */

    gtk_widget_show(colourseldlg);
}

return handled;
}

/* 关闭、退出事件处理函数 */

void destroy_window (GtkWidget *widget, gpointer client_data)
{
    gtk_main_quit ();
}

/* 主函数 */
gint main (gint argc, gchar *argv[])
{
    GtkWidget *window;

    /* 初始化GTK */
    gtk_init (&argc,&argv);

    /* 创建顶级窗口，设置标题，以及窗口是否可缩放 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW(window), "Color selection test");
    gtk_window_set_policy (GTK_WINDOW(window), TRUE, TRUE, TRUE);

    /* 为"delete"和"destroy"事件设置回调函数 */
    gtk_signal_connect (GTK_OBJECT(window), "delete_event",
        (GtkSignalFunc)destroy_window, (gpointer>window);

    gtk_signal_connect (GTK_OBJECT(window), "destroy",
        (GtkSignalFunc)destroy_window, (gpointer>window);

    /* 创建绘图区，设置尺寸，捕获鼠标按键事件 */
    drawingarea = gtk_drawing_area_new ();
```

```

gtk_drawing_area_size (GTK_DRAWING_AREA(drawingarea), 200, 200);
gtk_widget_set_events (drawingarea, GDK_BUTTON_PRESS_MASK);

gtk_signal_connect (GTK_OBJECT(drawingarea), "event",
    (GtkSignalFunc)area_event, (gpointer)drawingarea);

/* 将绘图区添加到窗口中，然后显示它们 */
gtk_container_add (GTK_CONTAINER(window), drawingarea);

gtk_widget_show (drawingarea);
gtk_widget_show (window);

/* 进入主循环 */
gtk_main ();
return(0);
}
/* 示例结束 */

```

上面代码的运行效果如图 9-8所示。在程序窗口上点击，就会弹出一个颜色选择对话框，选择的颜色会实时地显示在窗口中。

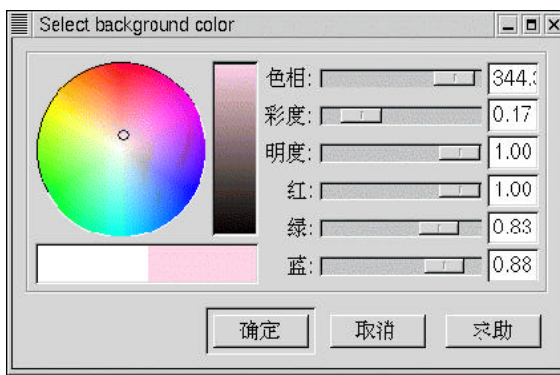


图9-8 颜色选择构件

9.13 文件选择构件GtkFileSelect

GtkFileSelect(文件选择构件)是一种快速、简单的显示文件对话框的方法。它带有“Ok”、“Cancel”、“Help”按钮，可以极大地减少编程时间。

可以用下面的方法创建文件选择构件：

```
GtkWidget *gtk_file_selection_new( gchar *title );
```

要设置文件名，例如，要在打开时指向指定目录，或者给定一个缺省文件名，可以使用下面的函数：

```

void gtk_file_selection_set_filename( GtkFileSelection *filesel,
    gchar *filename );

```

要获取用户输入或点击选中的文本，可以使用下面的函数：


```
gchar *gtk_file_selection_get_filename( GtkFileSelection *filesel );
```

还有几个指向文件选择构件内部的构件的指针，它们是：

dir_list

file_list

selection_entry

selection_text

main_vbox

ok_button

cancel_button

help_button

在为文件选择构件的信号设置回调函数时，极有可能用到 ok_button、cancel_button和 help_button三个指针。

下面的例子是来自 testgtk.c中的一段代码。在这个例子中，Help按钮出现在屏幕上，但是它什么也不做，因为没有为它的信号设置回调函数。

```
/* 文件选择构件示例开始 filesel.c */
```

```
#include <gtk/gtk.h>
```

```
/* 获得文件名，并将它打印到控制台上 */
```

```
void file_ok_sel (GtkWidget *w, GtkFileSelection *fs)
```

```
{
```

```
    g_print ("%s\n", gtk_file_selection_get_filename (
        GTK_FILE_SELECTION (fs)));
```

```
}
```

```
void destroy (GtkWidget *widget, gpointer data)
```

```
{
```

```
    gtk_main_quit ();
```

```
}
```

```
int main (int argc, char *argv[])
```

```
{
```

```
    GtkWidget *filew;
```

```
    gtk_init (&argc, &argv);
```

```
/* 创建一个新的文件选择构件 */
```

```
filew = gtk_file_selection_new ("File selection");
```

```
    gtk_signal_connect (GTK_OBJECT (filew), "destroy",
        (GtkSignalFunc) destroy, &filew);
```

```
/* 为ok_button按钮设置回调函数，连接到 file_ok_sel函数 */
```

```
gtk_signal_connect (GTK_OBJECT (GTK_FILE_SELECTION (filew)->ok_button),
    "clicked", (GtkSignalFunc) file_ok_sel, filew );
```

```
/* 为cancel_button设置回调函数，销毁构件 */
```

```
gtk_signal_connect_object (GTK_OBJECT (GTK_FILE_SELECTION
    (filew)->cancel_button),
```

```
"clicked", (GtkSignalFunc) gtk_widget_destroy,  
GTK_OBJECT (filew));  
  
/* 设置一个文件名，作为它的缺省文件名 */  
gtk_file_selection_set_filename (GTK_FILE_SELECTION(filew),  
                                "penguin.png");  
  
gtk_widget_show(filew);  
gtk_main ();  
return 0;  
}  
/* 示例结束 */
```

编译后，在shell提示符下输入./filesel，运行结果如图9-9所示。这个构件可以用于打开和保存文件。



图9-9 文件选择构件

第10章 容器构件GtkContainer

10.1 事件盒构件GtkEventBox

一些GTK构件没有与之相关联的X窗口，所以它们只在其父构件上显示其外观。由于这个原因，它们不能接收任何事件，并且，如果它们尺寸设置不正确，它们也不会自动剪裁，这样可能会把界面弄得乱糟糟的。如果要想构件接收事件，可以使用事件盒构件（GtkEventBox）。

初一看，GtkEventBox构件好像完全没有什么用。它在屏幕上什么也不画，并且对事件也不做响应。但是，它有一个功能：为它的子构件提供一个X窗口。因为许多GTK构件并没有相关联的X窗口，所以这一点很重要。虽然没有X窗口会节省内存，提高系统性能，但它也有一些弱点。没有X窗口的构件不能接收事件，并且对它的任何内容不实施剪裁。事件盒构件的名称强调它的事件处理功能，同时，它也能用于剪裁构件。

用以下函数创建一个新的事件盒构件：

```
GtkWidget *gtk_event_box_new( void );
```

然后子构件就可以添加到GtkEventBox里面：

```
gtk_container_add( GTK_CONTAINER(event_box), child_widget );
```

下面的示例演示了事件盒的用途：创建一个标签，将它剪裁，放到一个小盒子里面，然后设置让鼠标点击时程序退出。改变窗口的尺寸会使标签构件的尺寸发生变化。

```
/* 事件盒构件示例开始eventbox.c */
#include <gtk/gtk.h>
int
main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *event_box;
    GtkWidget *label;
    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Event Box");

    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (gtk_exit), NULL);

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个事件盒，然后将它加到顶级窗口上 */
    event_box = gtk_event_box_new ();
    gtk_container_add (GTK_CONTAINER(window), event_box);
    gtk_widget_show (event_box);

    /* 创建一个长标签 */
}
```

```

label = gtk_label_new ("Click here to quit, quit, quit, quit, quit");
gtk_container_add (GTK_CONTAINER (event_box), label);
gtk_widget_show (label);

/* 将标签剪裁短 */
gtk_widget_set_usize (label, 110, 20);

/* 为它绑定一个动作 */
gtk_widget_set_events (event_box, GDK_BUTTON_PRESS_MASK);
gtk_signal_connect (GTK_OBJECT(event_box), "button_press_event",
                    GTK_SIGNAL_FUNC (gtk_exit), NULL);

/* 为事件盒创建一个X窗口 */
gtk_widget_realize (event_box);
gdk_window_set_cursor (event_box->window, gdk_cursor_new (GDK_HAND1));
gtk_widget_show (window);
gtk_main ();
return(0);
}
/* 示例结束 */

```

将上面的代码保存为 eventbox.c，然后写一个下面这样的 Makefile：

```

CC = gcc
eventbox: eventbox.c
    $(CC) `gtk-config --cflags` eventbox.c -o eventbox `
    `gtk-config --libs`
clean:
    rm -f *.o eventbox

```

编译之后，运行结果如图 10-1 所示。改变窗口大小时，标签所显示的文本数量也会改变。点击标签，应用程序会退出。

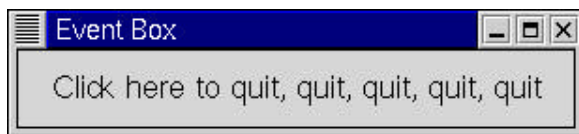


图10-1 使用事件盒，可以让标签构件对点击进行响应

10.2 对齐构件GtkAlignment

GtkAlignment(对齐构件)允许将一个构件放在相对于对齐构件窗口的某个位置和尺寸上。例如，将一个构件放在窗口的正中间时，就要使用对齐构件。

只有如下两个函数与对齐构件相关联：

```

GtkWidget* gtk_alignment_new( gfloat xalign,
                               gfloat yalign,
                               gfloat xscale,
                               gfloat yscale );

void gtk_alignment_set( GtkAlignment *alignment,

```

```

gfloat      xalign,
gfloat      yalign,
gfloat      xscale,
gfloat      yscale );

```

第一个函数用指定的参数创建新的对齐构件。第二个函数用于改变对齐构件的参数。

上面函数的所有四个参数都是介于0.0与1.0间的浮点数。xalign 和 yalign参数影响对齐构件内部的构件位置。xscale和yscale 参数影响分配给构件的空间总数。

使用以下函数可以将子构件添加到对齐构件中：

```
gtk_container_add( GTK_CONTAINER(alignment), child_widget );
```

要看关于对齐构件的例子，请看进度条构件的示例。

10.3 框架构件GtkFrame

GtkFrame(框架构件)可以用于在盒子中封装一个或一组构件，框架本身还可以有一个标签。标签的位置和风格可以灵活改变。

用下面函数可以创建新框架构件：

```
GtkWidget *gtk_frame_new( const gchar *label );
```

其中，label参数是框架构件的标签。

缺省设置时，标签放在框架的左上角。传递 NULL时，框架不显示标签。标签文本可以用下面的函数改变。

```

void gtk_frame_set_label( GtkFrame      *frame,
                          const gchar *label );

```

标签的位置可以用下面的函数改变：

```

void gtk_frame_set_label_align( GtkFrame *frame,
                                gfloat    xalign,
                                gfloat    yalign );

```

xalign和yalign参数取值范围介于0.0和1.0之间。xalign指定标签在框架构件上部水平线上的位置。yalign参数目前还没有使用。xalign的缺省值是0.0，它将标签放在框架构件的左上角处。

下面的函数用于改变frame的轮廓框的风格。

```

void gtk_frame_set_shadow_type( GtkFrame      *frame,
                                GtkShadowType type);

```

Type参数可以取以下值：

GTK_SHADOW_NONE

GTK_SHADOW_IN

GTK_SHADOW_OUT

GTK_SHADOW_ETCHED_IN (缺省值)

GTK_SHADOW_ETCHED_OUT

下面的例子演示了怎样使用框架构件。

```

/* 框架构件示例开始 frame.c */
#include <gtk/gtk.h>

```

```

int main( int    argc,
          char *argv[] )

```

```
{

/* 构件的存储类型是GtkWidget */
GtkWidget *window;
GtkWidget *frame;
GtkWidget *button;
gint i;

/* 初始化GTK */
gtk_init(&argc, &argv);
/* 创建一个新窗口 */
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(GTK_WINDOW(window), "Frame Example");

/* 为窗口的"destroy"事件设置一个回调函数 */
gtk_signal_connect (GTK_OBJECT (window), "destroy",
                    GTK_SIGNAL_FUNC (gtk_main_quit), NULL);

gtk_widget_set_usize(window, 300, 300);
/* 设置窗口的边框宽度 */
gtk_container_set_border_width (GTK_CONTAINER (window), 10);

/* 创建一个框架构件 */
frame = gtk_frame_new(NULL);
gtk_container_add(GTK_CONTAINER(window), frame);

/* 设置框架的标签 */
gtk_frame_set_label( GTK_FRAME(frame), "GTK Frame Widget" );

/* 将标签在框架构件的右边对齐 */
gtk_frame_set_label_align( GTK_FRAME(frame), 1.0, 0.0);

/* 设置框架构件的风格 */
gtk_frame_set_shadow_type( GTK_FRAME(frame), GTK_SHADOW_ETCHED_OUT);
gtk_widget_show(frame);

/* 显示窗口 */
gtk_widget_show (window);

/* 进入主循环 */
gtk_main ();

return(0);
}
/* 示例结束 */
```

将上述代码保存为frame.c, 然后编写一个像下面这样的Makefile文件。

```
CC = gcc
frame: frame.c
    $(CC) `gtk-config --cflags` frame.c -o frame \
```



```
`gtk-config --libs`
clean:
    rm -f *.o frame
```

运行结果如图 10-2 所示。请注意，框架构件的标题在构件的右上角。

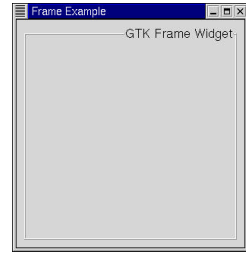


图10-2 框架构件

GtkAspectRatioFrame(比例框架构件)和框架构件差不多，差别在于它会保持子构件的长宽比例，如果需要，还会在构件中增加额外的可用空间。例如，想预览一个大的图片。当用户改变窗口的尺寸时，预览的尺寸会随之改变，但是纵横比例要与原来的尺寸保持一致。

用以下函数创建新比例框架构件：

```
GtkWidget *gtk_aspect_frame_new( const gchar *label,
                                gfloat          xalign,
                                gfloat          yalign,
                                gfloat          ratio,
                                gint            obey_child);
```

xalign和yalign参数指定对齐构件的调整值。如果 obey_child参数设置为TRUE，子构件的长宽比例会和它所请求的理想长宽比例相匹配。否则，比例值由 ratio参数指定。

用以下函数可以改变已有比例框架构件的选项：

```
void gtk_aspect_frame_set( GtkAspectRatioFrame *aspect_frame,
                           gfloat          xalign,
                           gfloat          yalign,
                           gfloat          ratio,
                           gint            obey_child);
```

在下面的示例中，程序用一个比例框架构件显示一个绘图区，纵横比例总是 2:1，而不管用户如何改变顶级窗口的尺寸。

```
/* 比例框架构件示例 开始aspectframe.c */
#include <gtk/gtk.h>
int
main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *aspect_frame;
    GtkWidget *drawing_area;
    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Aspect Frame");
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个比例框架构件，将它添加到顶级窗口中 */
    aspect_frame = gtk_aspect_frame_new("2:1" /* 标签*/
```

```

0.5, /* 方向居中 */
0.5, /* 方向居中 */
2, /* xsize/ysize = 2 */
FALSE /*忽略子构件的比例*/);

gtk_container_add (GTK_CONTAINER(window), aspect_frame);
gtk_widget_show (aspect_frame);

/* 添加一个子构件到比例框架构件中 */
drawing_area = gtk_drawing_area_new ();

/* 要求画一个200×200的窗口，但是比例框架构件会给出一个200×100
 * 的窗口，因为我们已经指定了2×1的比例值 */
gtk_widget_set_usize (drawing_area, 200, 200);
gtk_container_add (GTK_CONTAINER(aspect_frame), drawing_area);
gtk_widget_show (drawing_area);

gtk_widget_show (window);
gtk_main ();
return 0;
}
/* 示例结束 */

```

将上面的代码保存为 aspectframe.c，然后编写一个如下所示的 Makefile 文件。

```

CC = gcc
aspectframe: aspectframe.c
    $(CC) `gtk-config --cflags` aspectframe.c \
    -o aspectframe `gtk-config --libs`

clean:
    rm -f *.o aspectframe

```

编译后，执行结果见图 10-3。试着调整窗口的尺寸，看一看框架构件如何变化。

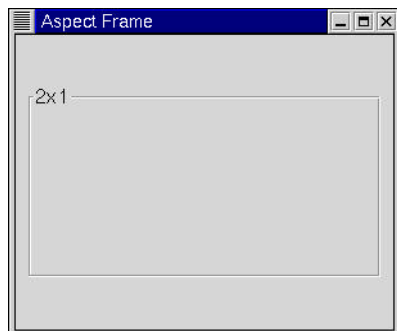


图10-3 比例框架构件

10.5 分栏窗口构件GtkPanedWindow

如果想要将一个窗口分成两个部分，可以使用 GtkPanedWindow(分栏窗口构件)。窗口两部分的尺寸由用户控制，它们之间有一个凹槽，上面有一个手柄，用户可以拖动此手柄改变

两部分的比。窗口划分可以是水平 (HPaned)或垂直的 (VPaned)。有两种分栏窗口构件：GtkHPaned (水平分栏窗口构件) 和 GtkVPaned (垂直分栏窗口构件)。

用以下函数创建新的分栏窗口构件：

```
GtkWidget *gtk_hpaned_new (void); 水平分栏窗口构件*/
```

```
GtkWidget *gtk_vpaned_new (void); 垂直分栏窗口构件*/
```

创建了分栏窗口构件后，可以在它的两边添加子构件。用下面的函数完成：

```
void gtk_paned_add1 (GtkPaned *paned,  
                    GtkWidget *child);
```

```
void gtk_paned_add2 (GtkPaned *paned,  
                    GtkWidget *child);
```

gtk_paned_add1() 将子构件添加到分栏窗口构件的左边或顶部。

gtk_paned_add2() 将子构件添加到分栏窗口构件的右边或下部。

如果希望在分栏窗口构件上设置很复杂的界面，可以将该构件和组盒 (GtkHBox和 GtkVBox) 或表格构件结合使用。

分栏窗口构件的视觉外观可以用以下函数改变：

```
void gtk_paned_set_handle_size( GtkPaned *paned,  
                                guint16   size);
```

```
void gtk_paned_set_gutter_size( GtkPaned *paned,  
                                guint16   size);
```

第一个函数设置手柄的尺寸，第二个函数设置两部分之间的凹槽的尺寸。

在下面的例子中，创建了一个假想的用户 Email程序的用户界面。窗口被垂直划分为两个部分，上面部分显示一个 Email信息列表，下部显示 Email文本信息。所有的程序都是相当直接的。有几点要注意：在文本构件显现前文本不能加到文本构件中，但你可以调用 gtk_widget_realize()函数完成，不过，作为一个技巧，我们可以为构件的“ realize ”信号设置一个信号处理函数，并在这个函数里面添加文本；还有，我们需要为包含文本窗口和它的滚动条的表格构件 (GtkTable) 设置GTK_SHRINK选项，以便当窗口的底部变小时，能够正确显示，而不是将下部的构件压到窗口的底部去。

```
/* 分栏窗口构件示例 paned.c */  
#include <gtk/gtk.h>  
  
/* 创建一个"信息"列表 */  
GtkWidget *  
create_list (void)  
{  
  
    GtkWidget *scrolled_window;  
    GtkWidget *list;  
    GtkWidget *list_item;  
  
    int i;  
    char buffer[16];  
    /* 创建一个新的分栏窗口构件，只有需要时，滚动条才出现 */  
    scrolled_window = gtk_scrolled_window_new (NULL, NULL);
```

```

gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC,
                                GTK_POLICY_AUTOMATIC);

/*创建一个新的列表框，将它放在分栏窗口构件中 */
list = gtk_list_new ();
gtk_scrolled_window_add_with_viewport (
    GTK_SCROLLED_WINDOW (scrolled_window), list);
gtk_widget_show (list);

/*在列表中添加一些列表项*/
for (i=0; i<10; i++) {
    sprintf(buffer, "Message #%d", i);
    list_item = gtk_list_item_new_with_label (buffer);
    gtk_container_add (GTK_CONTAINER(list), list_item);
    gtk_widget_show (list_item);
}

return scrolled_window;
}

/* 向文本构件中添加一些文本时必须先用一个回调函数实现文本构件。
 * 添加文本时，先将构件冻结，添加完成后，将构件解冻 */
void realize_text (GtkWidget *text, gpointer data)
{
    gtk_text_freeze (GTK_TEXT (text));
    gtk_text_insert (GTK_TEXT (text), NULL, &text->style->black, NULL,
        "From: pathfinder@nasa.gov\n"
        "To: mom@nasa.gov\n"
        "Subject: Made it!\n"
        "\n"
        "We just got in this morning. The weather has been\n"
        "great - clear but cold, and there are lots of fun sights.\n"
        "Sojourner says hi. See you soon.\n"
        " -Path\n", -1);

    gtk_text_thaw (GTK_TEXT (text));
}

/*创建一个滚动文本区域，用于显示添加的信息 */
GtkWidget *
create_text (void)
{
    GtkWidget *table;
    GtkWidget *text;
    GtkWidget *hscrollbar;
    GtkWidget *vscrollbar;

    /* 创建一个table构件，用于容纳文本构件和滚动条 */
    table = gtk_table_new (2, 2, FALSE);

```

```

/* 将文本构件放在table的左角，注意在
   * y方向设为GTK_SHRINK */
text = gtk_text_new (NULL, NULL);
gtk_table_attach (GTK_TABLE (table), text, 0, 1, 0, 1,
                  GTK_FILL | GTK_EXPAND,
                  GTK_FILL | GTK_EXPAND | GTK_SHRINK, 0, 0);
gtk_widget_show (text);

/* 将一个水平滚动条放在左下角 */
hscrollbar = gtk_hscrollbar_new (GTK_TEXT (text)->hadj);
gtk_table_attach (GTK_TABLE (table), hscrollbar, 0, 1, 1, 2,
                  GTK_EXPAND | GTK_FILL, GTK_FILL, 0, 0);
gtk_widget_show (hscrollbar);

/*将一个垂直滚动条放在右上角 */
vscrollbar = gtk_vscrollbar_new (GTK_TEXT (text)->vadj);
gtk_table_attach (GTK_TABLE (table), vscrollbar, 1, 2, 0, 1,
                  GTK_FILL, GTK_EXPAND | GTK_FILL | GTK_SHRINK, 0, 0);
gtk_widget_show (vscrollbar);

/*将一个处理函数添加到文本构件，当它实现时在文本构件上显示一段信息 */
gtk_signal_connect (GTK_OBJECT (text), "realize",
                   GTK_SIGNAL_FUNC (realize_text), NULL);

return table;
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *vpaned;
    GtkWidget *list;
    GtkWidget *text;

    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Paned Windows");
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                       GTK_SIGNAL_FUNC (gtk_main_quit), NULL);

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    gtk_widget_set_usize (GTK_WIDGET(window), 450, 400);
    /*在顶级窗口上添加一个垂直分栏窗口构件*/
    vpaned = gtk_vpaned_new ();
    gtk_container_add (GTK_CONTAINER(window), vpaned);
    gtk_paned_set_handle_size (GTK_PANED(vpaned),
                              10);
    gtk_paned_set_gutter_size (GTK_PANED(vpaned),
                              15);
    gtk_widget_show (vpaned);

    /*在分格窗口的两边添加一些构件*/

```

```

list = create_list ();
gtk_paned_add1 (GTK_PANED(vpaned), list);
gtk_widget_show (list);
    text = create_text ();
    gtk_paned_add2 (GTK_PANED(vpaned), text);
    gtk_widget_show (text);
    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
/* 示例结束 */

```

编译后的运行结果如图 10-4所示。尝试改变窗口尺寸，观察上下两部分是怎么变的，再试着改变一下上下两部分的相对比例，看看其效果。

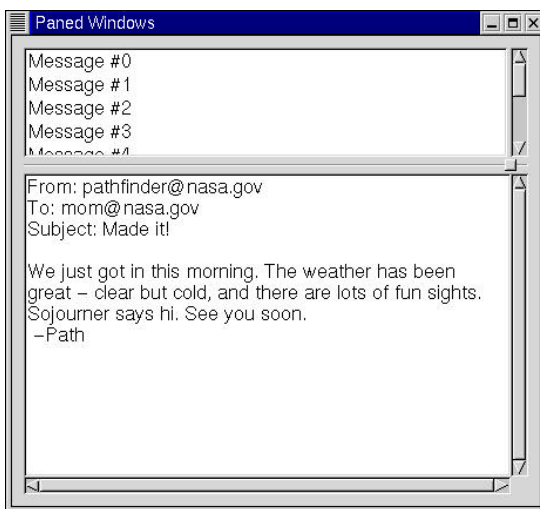


图10-4 分栏窗口示例

10.6 视角构件GtkViewport

一般很少直接使用 GtkViewport(视角构件)。多数情况下是使用分栏窗口构件，由分栏窗口构件使用视角构件。

视角构件允许在其中放置一个较大的构件，这样可以一次只看到构件的一部分。它用调整对象定义要显示的区域。

用以下函数创建视角构件。

```

GtkWidget *gtk_viewport_new( GtkAdjustment *hadjustment,
                             GtkAdjustment *vadjustment );

```

可以看到，创建构件时能够指定构件使用的水平和垂直调整对象。如果给函数传递一个 NULL 参数，构件会自己创建一个调整对象。

创建构件后，可以用以下函数设置和取得它的调整对象：

```

GtkAdjustment *gtk_viewport_get_hadjustment (GtkViewport *viewport );
GtkAdjustment *gtk_viewport_get_vadjustment (GtkViewport *viewport );
void gtk_viewport_set_hadjustment( GtkViewport *viewport,

```

```

                                GtkAdjustment *adjustment );
void gtk_viewport_set_vadjustment( GtkWidget      *viewport,
                                GtkAdjustment *adjustment );

```

下面的函数可用于改变视角构件的外观：

```

void gtk_viewport_set_shadow_type( GtkWidget      *viewport,
                                GtkShadowType type );

```

Type参数可以取以下值：

```

GTK_SHADOW_NONE,
GTK_SHADOW_IN,
GTK_SHADOW_OUT,
GTK_SHADOW_ETCHED_IN,
GTK_SHADOW_ETCHED_OUT

```

10.7 滚动窗口构件GtkScrolledWindow

GtkScrolledWindow(滚动窗口构件)用于创建一个可滚动区域，并将其他构件放入其中。可以在滚动窗口中插入任何其他构件，在其内部的构件不论尺寸大小都可以通过滚动条访问到。

用下面的函数创建新的滚动窗口构件。

```

GtkWidget *gtk_scrolled_window_new( GtkAdjustment *hadjustment,
                                GtkAdjustment *vadjustment );

```

第一个参数是水平方向的调整对象，第二个参数是垂直方向的调整对象。它们总是设置为NULL。

```

void gtk_scrolled_window_set_policy( GtkScrolledWindow *swindowed_
                                GtkPolicyType      hscrollbar_policy,
                                GtkPolicyType      vscrollbar_policy );

```

这个函数可以设置滚动条出现的方式。第一个参数是要设置的滚动窗口构件，第二个设置水平滚动条出现的方式，第三个参数设置垂直滚动条的方式。

滚动条的方式取值可以为GTK_POLICY_AUTOMATIC或GTK_POLICY_ALWAYS。当要求滚动条根据需要自动出现时，可设为GTK_POLICY_AUTOMATIC；若设为GTK_POLICY_ALWAYS，滚动条会一直出现在滚动窗口构件上。

可以用以下函数将构件放在滚动窗口构件上：

```

void gtk_scrolled_window_add_with_viewport(
GtkWidget *scrolled_window,
GtkWidget *child);

```

下面是一个简单例子：在滚动窗口构件中放置一个表格构件，并在表格中放 100个开关按钮。

```

/* 滚动窗口示例开始 scrolledwin.c */
#include <gtk/gtk.h>
void destroy(GtkWidget *widget, gpointer data)
{
    gtk_main_quit();
}

```

```
int main (int argc, char *argv[])
{
    static GtkWidget *window;
    GtkWidget *scrolled_window;
    GtkWidget *table;
    GtkWidget *button;
    char buffer[32];
    int i, j;

    gtk_init (&argc, &argv);

    /* 创建一个新的对话框构件，滚动窗口构件就放在这个窗口上 */
    window = gtk_dialog_new ();
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        (GtkSignalFunc) destroy, NULL);
    gtk_window_set_title (GTK_WINDOW (window), "GtkScrolledWindow example");
    gtk_container_set_border_width (GTK_CONTAINER (window), 0);
    gtk_widget_set_usize(window, 300, 300);
    /* 创建一个新的滚动窗口构件 */
    scrolled_window = gtk_scrolled_window_new (NULL, NULL);
    gtk_container_set_border_width (GTK_CONTAINER (scrolled_window), 10);

    /* 滚动条的出现方式可以是GTK_POLICY_AUTOMATIC或GTK_POLICY_ALWAYS。
    * 设为GTK_POLICY_AUTOMATIC将自动决定是否出现滚动条
    * 而设为GTK_POLICY_ALWAYS，将一直显示一个滚动条
    * 第一个是水平滚动条，第二个是垂直滚动条
    * */
    gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                    GTK_POLICY_AUTOMATIC,
                                    GTK_POLICY_ALWAYS);

    /* 对话框窗口内部包含一个vbox构件*/
    gtk_box_pack_start (GTK_BOX (GTK_DIALOG(window)->vbox), scrolled_window,
                        TRUE, TRUE, 0);
    gtk_widget_show (scrolled_window);

    /* 创建一个包含10×10个方格的表格GtkTable */
    table = gtk_table_new (10, 10, FALSE);

    /* 设置x和y方向的行间距为10像素 */
    gtk_table_set_row_spacings (GTK_TABLE (table), 10);
    gtk_table_set_col_spacings (GTK_TABLE (table), 10);

    /* 将表格组装到滚动窗口构件中 */
    gtk_scrolled_window_add_with_viewport (
        GTK_SCROLLED_WINDOW (scrolled_window), table);
    gtk_widget_show (table);

    /* 在表格中添加切换按钮以展示滚动窗口构件 */
    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++) {
            sprintf (buffer, "button (%d,%d)\n", i, j);
```



```

button = gtk_toggle_button_new_with_label (buffer);
gtk_table_attach_defaults (GTK_TABLE (table), button,
                           i, i+1, j, j+1);
gtk_widget_show (button);
}

/* 在对话框的底部添加一个 "close"按钮 */
button = gtk_button_new_with_label ("close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           (GtkSignalFunc) gtk_widget_destroy,
                           GTK_OBJECT (window));

/* 让按钮成为缺省按钮 */

GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area),
                    button, TRUE, TRUE, 0);

/* 让按钮固定为缺省按钮, 只要按回车键就相当于点击了这个按钮 */
gtk_widget_grab_default (button);
gtk_widget_show (button);
gtk_widget_show (window);
gtk_main();
return(0);
}
/* 示例结束 */

```

编译后, 示例的运行结果如图 10-5所示。尝试改变窗口的大小, 可以看到滚动条是如何起作用的。还可以用 `gtk_widget_set_usize()` 函数设置窗口或构件的缺省尺寸。

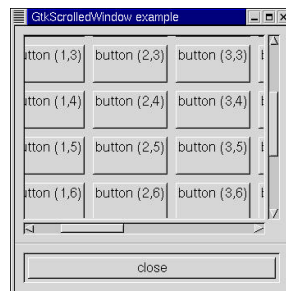


图10-5 滚动窗口构件

10.8 按钮盒构件GtkButtonBox

GtkButtonBox(按钮盒构件)可以很方便地快速布置一组按钮。它有水平和垂直两种样式。你可以用以下函数创建垂直或水平按钮盒：

```

GtkWidget *gtk_hbutton_box_new( void );
GtkWidget *gtk_vbutton_box_new( void );

```

与按钮盒相关的唯一属性是按钮如何放置。可以用以下函数改变按钮间的间距：

```

void gtk_hbutton_box_set_spacing_default( gint spacing );
void gtk_vbutton_box_set_spacing_default( gint spacing );

```

与此相似，可以用下面的函数取得当前的间距值：

```

gint gtk_hbutton_box_get_spacing_default( void );
gint gtk_vbutton_box_get_spacing_default( void );

```

我们能访问的第二个属性会影响按钮在按钮盒中的布局。可以用下面的函数设置它：

```

void gtk_hbutton_box_set_layout_default( GtkButtonBoxLayout layout );
void gtk_vbutton_box_set_layout_default( GtkButtonBoxLayout layout );

```

layout参数可以取以下值：

GTK_BUTTONBOX_DEFAULT_STYLE

GTK_BUTTONBOX_SPREAD

GTK_BUTTONBOX_EDGE

GTK_BUTTONBOX_START

GTK_BUTTONBOX_END

当前的布局设置可以用以下函数取得：

```
GtkButtonBoxStyle gtk_hbutton_box_get_layout_default( void );
GtkButtonBoxStyle gtk_vbutton_box_get_layout_default( void );
```

用以下函数可以将按钮添加到按钮盒中：

```
gtk_container_add( GTK_CONTAINER(button_box), child_widget );
```

下面的例子演示了按钮盒的不同布局设置。

```
/* 按钮盒示例开始buttonbox.c */
#include <gtk/gtk.h>
```

```
/* 用指定的参数创建一个按钮盒 */
```

```
GtkWidget *create_bbox (gint horizontal,
                        char* title,
                        gint spacing,
                        gint child_w,
                        gint child_h,
                        gint layout)
{
    GtkWidget *frame;
    GtkWidget *bbox;
    GtkWidget *button;

    frame = gtk_frame_new (title);

    if (horizontal)
        bbox = gtk_hbutton_box_new ();
    else
        bbox = gtk_vbutton_box_new ();
    gtk_container_set_border_width (GTK_CONTAINER (bbox), 5);
    gtk_container_add (GTK_CONTAINER (frame), bbox);

    /* 设置按钮盒的外观 */
    gtk_button_box_set_layout (GTK_BUTTON_BOX (bbox), layout);
    gtk_button_box_set_spacing (GTK_BUTTON_BOX (bbox), spacing);
    gtk_button_box_set_child_size (GTK_BUTTON_BOX (bbox), child_w, child_h);

    button = gtk_button_new_with_label ("OK");
    gtk_container_add (GTK_CONTAINER (bbox), button);

    button = gtk_button_new_with_label ("Cancel");
    gtk_container_add (GTK_CONTAINER (bbox), button);

    button = gtk_button_new_with_label ("Help");
    gtk_container_add (GTK_CONTAINER (bbox), button);
    return(frame);
}
```

```
}

int main( int    argc,
          char *argv[] )
{
    static GtkWidget* window = NULL;
    GtkWidget *main_vbox;
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *frame_horz;
    GtkWidget *frame_vert;

    /* 初始化GTK */
    gtk_init( &argc, &argv );
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Button Boxes");

    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC(gtk_main_quit),
                        NULL);

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    main_vbox = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), main_vbox);

    frame_horz = gtk_frame_new ("Horizontal Button Boxes");
    gtk_box_pack_start (GTK_BOX (main_vbox), frame_horz, TRUE, TRUE, 10);

    vbox = gtk_vbox_new (FALSE, 0);
    gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);
    gtk_container_add (GTK_CONTAINER (frame_horz), vbox);

    gtk_box_pack_start (GTK_BOX (vbox),
                        create_bbox (TRUE, "Spread (spacing 40)",
                                    40, 85, 20, GTK_BUTTONBOX_SPREAD),
                        TRUE, TRUE, 0);
    gtk_box_pack_start (GTK_BOX (vbox),
                        create_bbox (TRUE, "Edge (spacing 30)",
                                    30, 85, 20, GTK_BUTTONBOX_EDGE),
                        TRUE, TRUE, 5);

    gtk_box_pack_start (GTK_BOX (vbox),
                        create_bbox (TRUE, "Start (spacing 20)",
                                    20, 85, 20, GTK_BUTTONBOX_START),
                        TRUE, TRUE, 5);

    gtk_box_pack_start (GTK_BOX (vbox),
                        create_bbox (TRUE, "End (spacing 10)",
                                    10, 85, 20, GTK_BUTTONBOX_END),
                        TRUE, TRUE, 5);
}
```

```

frame_vert = gtk_frame_new ("Vertical Button Boxes");
gtk_box_pack_start (GTK_BOX (main_vbox), frame_vert, TRUE, TRUE, 10);
hbox = gtk_hbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (hbox), 10);
gtk_container_add (GTK_CONTAINER (frame_vert), hbox);

gtk_box_pack_start (GTK_BOX (hbox),
    create_bbox (FALSE, "Spread (spacing 5)",
        5, 85, 20, GTK_BUTTONBOX_SPREAD),
    TRUE, TRUE, 0);

gtk_box_pack_start (GTK_BOX (hbox),
    create_bbox (FALSE, "Edge (spacing 30)",
        30, 85, 20, GTK_BUTTONBOX_EDGE),
    TRUE, TRUE, 5);

gtk_box_pack_start (GTK_BOX (hbox),
    create_bbox (FALSE, "Start (spacing 20)",
        20, 85, 20, GTK_BUTTONBOX_START),
    TRUE, TRUE, 5);

gtk_box_pack_start (GTK_BOX (hbox),
    create_bbox (FALSE, "End (spacing 20)",
        20, 85, 20, GTK_BUTTONBOX_END),
    TRUE, TRUE, 5);

gtk_widget_show_all (window);

/* 进入事件循环 */
gtk_main ();
return(0);
}
/* 示例结束 */

```

以上代码的执行效果如图 10-6 所示。其中演示了按钮盒的各种组织方式。

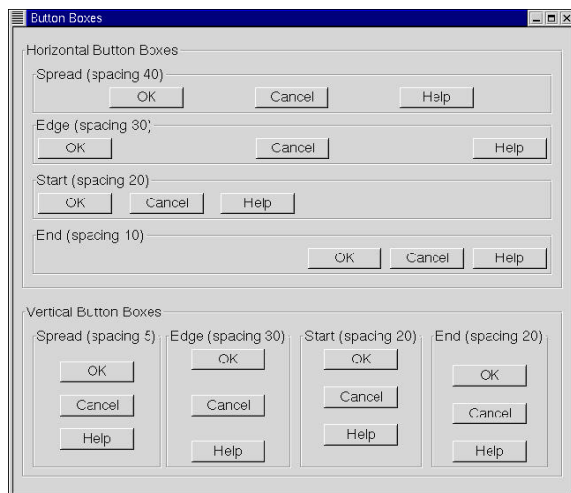


图10-6 按钮盒

10.9 工具条构件GtkToolbar

GtkToolbar(工具条构件)是非常常见的构件。它实际上是一个容器,你在上面可以添加各种各样的其他构件。最常见的情况是在上面添加普通按钮构件、开关按钮构件、无线按钮构件等。当然,也可以添加GtkTree等构件。不过,这样可能会使工具条看起来有点古怪。

应用程序的图形用户接口一般都会向同一个功能或者操作提供多种访问方式。例如,打开文件的功能可以通过“文件/打开”菜单项完成,或者通过快捷键Ctrl+O(同时按下Ctrl键和字母O键),或者通过点击工具条上的“打开”按钮。工具条将一些构件分组,这样能够简化定制它们的外观和布局。典型情况下工具条由带图标和标签以及工具提示的按钮组成,不过,其他构件也可以放在工具条里面。按钮可以水平或垂直排列,还可以显示图标或标签,或者两者都显示。

一般情况下,在普通GtkWindow构件上创建的工具条是不能浮动的,也就是说,它不能脱离窗口。如果使用GnomeApp构件作为应用程序的主窗口,在GnomeApp上就以一个GtkHandleBox作为容器容纳工具条,这样创建的工具条可以拖动离开主窗口,浮动在桌面上,可以给人非常新颖的感觉。另外,用下面的方法创建工具条比较复杂。Gnome库提供了一种GnomeUIInfo模板,用于创建工具条和菜单非常方便。请参看后面的相关章节。

用以下函数创建工具条:

```
GtkWidget *gtk_toolbar_new( GtkOrientation orientation,
                             GtkToolbarStyle style );
```

Orientation参数可以取以下值:

GTK_ORIENTATION_HORIZONTAL

GTK_ORIENTATION_VERTICAL

Style参数可以取以下值:

GTK_TOOLBAR_TEXT

GTK_TOOLBAR_ICONS

GTK_TOOLBAR_BOTH

style参数适用于所有用与“item”相关的函数创建的按钮(不包括作为分离构件插入到工具条的按钮)。

创建工具条以后,可以向其中追加、前插和插入工具条项(这里意指简单文本字符串或元素(这里意指任何构件类型)。要想描述一个工具条上的对象,需要一个标签文本、一个工具提示文本、一个私有工具提示文本、一个图标和一个回调函数。例如,要前插或追加一个按钮,应该使用下面的函数。

向工具条上追加(添加在其他所有构件的后面)构件:

```
GtkWidget *gtk_toolbar_append_item( GtkToolbar toolbar,
                                     const char *text,
                                     const char *tooltip_text,
                                     const char *tooltip_private_text,
                                     GtkWidget *icon,
                                     GtkSignalFunc callback,
                                     gpointer user_data );
```

向工具条上前插(添加在其他所有构件的前面)构件:

```

GtkWidget *gtk_toolbar_prepend_item( GtkWidget      *toolbar,
                                   const char      *text,
                                   const char      *tooltip_text,
                                   const char      *tooltip_private_text,
                                   GtkWidget      *icon,
                                   GtkSignalFunc   callback,
                                   gpointer        user_data );

```

如果要使用`gtk_toolbar_insert_item`函数，除上面函数中要指定的参数以外，还要指定插入对象的位置，形式如下：

```

GtkWidget *gtk_toolbar_insert_item( GtkWidget      *toolbar,
                                   const char      *text,
                                   const char      *tooltip_text,
                                   const char      *tooltip_private_text,
                                   GtkWidget      *icon,
                                   GtkSignalFunc   callback,
                                   gpointer        user_data,
                                   gint            position );

```

要简化在工具条项之间添加空白区域，可以使用下面的函数。

在工具条上追加空白区域：

```
void gtk_toolbar_append_space( GtkWidget *toolbar );
```

在工具条上前插空白区域：

```
void gtk_toolbar_prepend_space( GtkWidget *toolbar );
```

在工具条上的`position`位置插入空白区域：

```
void gtk_toolbar_insert_space( GtkWidget *toolbar,
                              gint        position );
```

可以用下面的函数设置整个工具条上的空白区域的大小：

```
void gtk_toolbar_set_space_size( GtkWidget *toolbar,
                                gint        space_size );
```

如果需要，工具条的放置方向和它的式样可以在运行时用下面的函数设置。

设置工具条的放置方向：

```
void gtk_toolbar_set_orientation( GtkWidget      *toolbar,
                                 GtkOrientation   orientation );
```

设置工具条的样式：

```
void gtk_toolbar_set_style( GtkWidget      *toolbar,
                           GtkToolbarStyle style );
```

启用或者禁用工具条的工具提示文本：

```
void gtk_toolbar_set_tooltips( GtkWidget *toolbar,
                              gint        enable );
```

其中，`enable`设为`TRUE`则启用工具提示文本，否则，禁止显式工具提示文本。

上面第一个函数中的`orientation`参数是枚举值，可以取`GTK_ORIENTATION_HORIZONTAL`或`GTK_ORIENTATION_VERTICAL`。`style`参数用于设置工具条项的外观，可以取`GTK_TOOLBAR_ICONS`（只显示图标），`GTK_TOOLBAR_TEXT`（只显示文本）或`GTK_TOOLBAR_BOTH`（同时显示图标和文本）。

要想了解工具条能做什么，看一看下面的程序（在代码之间我们插入了一些解释）：

```
#include <gtk/gtk.h>
#include "gtk.xpm"
/* 这个函数连接到 "close"按钮或者从窗口管理器关闭窗口的事件上 */
void delete_event (GtkWidget *widget, GdkEvent *event, gpointer data)
{
    gtk_main_quit ();
}
```

上面的代码和其他的 GTK应用程序差别不大，有一点不同的是：我们包含了一个漂亮的 XPM图片，用作所有按钮的图标。

```
GtkWidget* close_button; 按钮，引发一个信号以关闭应用程序 */
GtkWidget* tooltips_button; 启用/禁用工具提示 */
GtkWidget* text_button,
    * icon_button,
    * both_button; /*无线按钮 */
GtkWidget* entry; /*个文本输入构件，用于显示组装到工具条上的构件 */
```

事实上，不是上面所有的构件都是必须的，我把它们放在一起，是为了让事情更清晰。

```
/* 当按钮进行状态切换时，我们检查哪一个按钮是活动的，依此设置工具条的样式
 * 注意，工具条是作为用户数据传递到回调函数的 */
void radio_event (GtkWidget *widget, gpointer data)
{
    if (GTK_TOGGLE_BUTTON (text_button)->active)
        gtk_toolbar_set_style(GTK_TOOLBAR ( data ), GTK_TOOLBAR_TEXT);
    else if (GTK_TOGGLE_BUTTON (icon_button)->active)
        gtk_toolbar_set_style(GTK_TOOLBAR ( data ), GTK_TOOLBAR_ICONS);
    else if (GTK_TOGGLE_BUTTON (both_button)->active)
        gtk_toolbar_set_style(GTK_TOOLBAR ( data ), GTK_TOOLBAR_BOTH);
}
```

```
/* 检查给定开关按钮的状态，依此启用或禁用工具提示 */
void toggle_event (GtkWidget *widget, gpointer data)
{
    gtk_toolbar_set_tooltips (GTK_TOOLBAR ( data ),
                               GTK_TOGGLE_BUTTON (widget)->active );
}
```

上面是当工具条上的一个按钮被按下时要调用的两个回调函数。

```
int main (int argc, char *argv[])
{
    /* 下面创建主窗口（一个对话框）和一个 GtkHandle构件*/
    GtkWidget* dialog;
    GtkWidget* handlebox;
    GtkWidget * toolbar;
    GdkPixmap * icon;
    GdkBitmap * mask;
    GtkWidget * iconw;

    /* 初始化GTK */
    gtk_init (&argc, &argv);

    /* 用给定的标题和尺寸创建一个新窗口 */
```

```

dialog = gtk_dialog_new ();
gtk_window_set_title ( GTK_WINDOW ( dialog ) , "GTKToolbar Tutorial");
gtk_widget_set_usize( GTK_WIDGET ( dialog ) , 600 , 300 );
GTK_WINDOW ( dialog ) ->allow_shrink = TRUE;

/* 在关闭窗口时退出 */
gtk_signal_connect ( GTK_OBJECT ( dialog ) , "delete_event",
                    GTK_SIGNAL_FUNC ( delete_event ), NULL);

/* 需要显现窗口，因为我们要在它的环境中为工具条设置图片 */
gtk_widget_realize ( dialog );

/* 我们将工具条放在一个手柄构件 ( GtkHandle ) 上，
 * 这样它可以从主窗口上移开 */
handlebox = gtk_handle_box_new ();
gtk_box_pack_start ( GTK_BOX ( GTK_DIALOG(dialog)->vbox ) ,
                    handlebox, FALSE, FALSE, 5 );

```

上面的代码和任何其他 Gtk应用程序都差不多。它们进行 GTK初始化，创建主窗口等。唯一需要解释的是：GtkHandlebox(手柄盒构件)。手柄盒构件只是一个可以在其中组装构件的盒子。它和普通盒子的区别在于它能从一个父窗口移开（事实上，手柄盒构件保留在父窗口上，但是它缩小为一个非常小的矩形，同时它的所有内容重新放在一个新的可自由移动的浮动窗口上）。拥有一个可浮动工具条给人感觉非常好，所以这两种构件经常同时使用。

```

/* 工具条设置为水平的，同时带有图标和文本
 * 在每个项之间有5像素的间距，
 * 并且，我们将它们全部放在手柄盒构件上 */
toolbar = gtk_toolbar_new ( GTK_ORIENTATION_HORIZONTAL,
                           GTK_TOOLBAR_BOTH );

gtk_container_set_border_width ( GTK_CONTAINER ( toolbar ) , 5 );
gtk_toolbar_set_space_size ( GTK_TOOLBAR ( toolbar ) , 5 );
gtk_container_add ( GTK_CONTAINER ( handlebox ) , toolbar );

icon = gdk_pixmap_create_from_xpm_d ( dialog->window, &mask,
                                     &dialog->style->white, gtk_xpm );

```

上面的代码初始化工具条，并创建了一个 GDKPixmap图片。

```

/* 工具条上第一项是"close"按钮 */
iconw = gtk_pixmap_new ( icon, mask )图标构件*/
close_button =
gtk_toolbar_append_item ( GTK_TOOLBAR ( toolbar工具条*/
                        "Close",           按钮标签 */
                        "Closes this app", 按钮的工具提示 */
                        "Private",         工具提示私有信息 */
                        iconw,             图标构件 */
                        GTK_SIGNAL_FUNC ( delete_event ),一个信号 */
                        NULL );

gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar )按钮之间的空白 */

```

在上面的代码中，可以看到最简单的情况：在工具条上增加一个按钮。在追加一个新的工具条项前，必须构造一个 pixmap构件用作该项的图标，这个步骤我们要对每一个工具条项

重复一次。在工具条项之间还要增加分隔空间，这样后面的工具条项就不会一个接一个紧挨着。可以看到，`gtk_toolbar_append_item`返回一个指向新创建的按钮构件的指针，所以我们可以用正常的方式使用它。

```
/* 现在，我们创建无线按钮组 */
iconw = gtk_pixmap_new ( icon, mask );
icon_button = gtk_toolbar_append_element(
    GTK_TOOLBAR(toolbar),
    GTK_TOOLBAR_CHILD_RADIOBUTTON, 元素类型 */
    NULL,                            指向构件的指针 */
    "Icon",                          标签 */
    "Only icons in toolbar",         工具提示 */
    "Private",                       工具提示私有字符串 */
    iconw,                           图标 */
    GTK_SIGNAL_FUNC (radio_event), 信号 */
    toolbar);                        信号传递的数据 */
gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ) );
```

这里我们通过使用 `gtk_toolbar_append_element` 函数创建了一个无线按钮组。事实上，使用这个函数，我们能够添加简单的工具条项，或者设置添加按钮间的分隔条（`type = GTK_TOOLBAR_CHILD_SPACE` 或 `GTK_TOOLBAR_CHILD_BUTTON`）。在上面的例子中，我们先创建了一个无线按钮组。要为此组创建其他按钮，需要一个指向前一个按钮的指针，以便可以很容易地创建一系列按钮。

```
/* 下面引用的无线按钮就是前面创建的按钮 */
iconw = gtk_pixmap_new ( icon, mask );
text_button =
    gtk_toolbar_append_element(GTK_TOOLBAR(toolbar),
                              GTK_TOOLBAR_CHILD_RADIOBUTTON,
                              icon_button,
                              "Text",
                              "Only texts in toolbar",
                              "Private",
                              iconw,
                              GTK_SIGNAL_FUNC (radio_event),
                              toolbar);

gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ) );

iconw = gtk_pixmap_new ( icon, mask );
both_button =
    gtk_toolbar_append_element(GTK_TOOLBAR(toolbar),
                              GTK_TOOLBAR_CHILD_RADIOBUTTON,
                              text_button,
                              "Both",
                              "Icons and text in toolbar",
                              "Private",
                              iconw,
                              GTK_SIGNAL_FUNC (radio_event),
                              toolbar);

gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ) );
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(both_button), TRUE);
```

最后，我们必须手工设置其中一个按钮的状态（否则它们全部保留为活动状态，并阻止我们在它们之间进行状态切换）。

```
/* 下面只是一个简单的开关按钮 */
iconw = gtk_pixmap_new ( icon, mask );
tooltips_button =
    gtk_toolbar_append_element(GTK_TOOLBAR(toolbar),
                              GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
                              NULL,
                              "Tooltips",
                              "Toolbar with or without tips",
                              "Private",
                              iconw,
                              GTK_SIGNAL_FUNC (toggle_event),
                              toolbar);

gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ) );
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(tooltips_button),TRUE);

/* 要将一个构件组装到工具条上，只需创建它，然后将它追加
 * 加到工具条上，同时设置合适的工具提示 */
entry = gtk_entry_new ();
gtk_toolbar_append_widget( GTK_TOOLBAR (toolbar),
                           entry,
                           "This is just an entry",
                           "Private" );

/* 因为它不是在工具条上创建的，所以我们必须显示它 */
gtk_widget_show ( entry );
```

可以看到，将任何构件添加到工具条上都是非常简单的。唯一要记住的是，这个构件必须手工显示(与此相反，在工具条内创建的工具条项随工具条一起显示)。

```
/* 好了，现在可以显示所有的东西了 */
gtk_widget_show ( toolbar );
gtk_widget_show ( handlebox );
gtk_widget_show ( dialog );

/* 进入主循环，等待用户的操作 */
gtk_main ();

return 0;
}
```

上面就是工具条教程的全部代码。当然，还需要一个漂亮的 XPM图片。下面就是一个：

```
/* XPM */
static char * gtk_xpm[] = {
"32 39 5 1",
".      c none",
"+      c black",
"@      c #3070E0",
"#      c #F05050",
"$      c #35E035",
```

10.10 笔记本构件GtkNotebook

一旦创建了笔记本构件，就可以使用一系列的函数操作该构件。下面将对它们进行分别

讨论。

先看一下怎样定位页面指示器——或称页标签，可以有四种位置：

```
void gtk_notebook_set_tab_pos( GtkNotebook      *notebook,
                               GtkPositionType  pos );
```

GtkPositionType参数可以取以下几个值，从字面上很容易理解它们的含义：

GTK_POS_LEFT	将标签页放在左边
GTK_POS_RIGHT	将标签页放在右边
GTK_POS_TOP	将标签页放在顶部
GTK_POS_BOTTOM	将标签页放在底部

它的缺省值是GTK_POS_TOP。

下面看一下怎样向笔记本中添加页面。有三种方法向笔记本中添加页面。前两种方法是非常相似的。

在笔记本构件中追加页面：

```
void gtk_notebook_append_page( GtkNotebook *notebook,
                               GtkWidget    *child,
                               GtkWidget    *tab_label );
```

在笔记本中前插页面：

```
void gtk_notebook_prepend_page( GtkNotebook *notebook,
                                GtkWidget    *child,
                                GtkWidget    *tab_label );
```

child 参数是放在笔记本上的子构件，tab_label是要添加的页面的标签。子构件必须分别创建，一般是一个容器构件，比如说表格构件。

最后一个函数与前两个函数类似，不过允许指定页面加入的位置。

```
void gtk_notebook_insert_page( GtkNotebook *notebook,
                               GtkWidget    *child,
                               GtkWidget    *tab_label,
                               gint         position );
```

其中的参数与_append_ and _prepend_函数一样，还包含一个额外参数——插入位置。该参数指定页面应该插入到哪一页。注意，第一页位置为0。

前面介绍了怎样添加页面，下面介绍怎样删除页面。

```
void gtk_notebook_remove_page( GtkNotebook *notebook,
                               gint         page_num );
```

这个函数从笔记本中删除由page_num参数指定的页面。

用以下函数寻找笔记本构件的当前标签页：

```
gint gtk_notebook_get_current_page( GtkNotebook *notebook );
```

下面两个函数将笔记本构件的页面向前或向后移动。对要操作的笔记本构件使用以下函数就可以了。

注意 当笔记本构件在最后一页时，调用gtk_notebook_next_page 函数，笔记本构件会跳到第一页。同样，如果笔记本构件在第一页，调用了函数 gtk_notebook_prev_page，笔记本构件会跳到最后一页。

```
void gtk_notebook_next_page( GtkNoteBook *notebook );
```

```
void gtk_notebook_prev_page( GtkNoteBook *notebook );
```

下面的函数设置“活动”页面。缺省状态下，笔记本显示第一页。

```
void gtk_notebook_set_page( GtkNotebook *notebook,
                             gint         page_num );
```

下面的函数显示或隐藏笔记本构件的页标签以及它的边框。

```
void gtk_notebook_set_show_tabs( GtkNotebook *notebook,
                                   gboolean     show_tabs);
void gtk_notebook_set_show_border( GtkNotebook *notebook,
                                    gboolean     show_border );
```

第一个函数中的show_tabs为TRUE则显示页标签，FALSE不显示页标签。

第二个函数中的show_border为TRUE则显示边框，为FALSE不显示边框。

如果页面较多，标签页在页面上排列不下时，可以用下面的函数。它允许标签页用两个按钮箭头滚动。

```
void gtk_notebook_set_scrollable( GtkNotebook *notebook,
                                   gboolean     scrollable );
```

show_tabs, show_border 和scrollable参数可以是TRUE或FALSE。

下面是一个关于GtkBooknote构件的示例。这个程序创建了一个带一个笔记本构件和 6个按钮的窗口。笔记本构件包含 11页，你可以用三种方式对它们进行添加：追加、前插、插入。点击按钮可以改变标签的位置，添加、删除标签和边框，删除一页，向前或向后改变标签页，以及退出程序。

```
/* 笔记本构件示例开始notebook.c */

#include <gtk/gtk.h>

/* 这个函数旋转标签页的位置 */
void rotate_book (GtkButton *button, GtkNotebook *notebook)
{
    gtk_notebook_set_tab_pos (notebook, (notebook->tab_pos +1) %4);
}

/* 添加/删除页标签和边框*/
void tabsborder_book (GtkButton *button, GtkNotebook *notebook)
{
    gint tval = FALSE;
    gint bval = FALSE;
    if (notebook->show_tabs == 0)
        tval = TRUE;
    if (notebook->show_border == 0)
        bval = TRUE;

    gtk_notebook_set_show_tabs (notebook, tval);
    gtk_notebook_set_show_border (notebook, bval);
}

/* 从笔记本构件上删除页面 */
void remove_book (GtkButton *button, GtkNotebook *notebook)
{

```

```
gint page;
page = gtk_notebook_get_current_page(notebook);
gtk_notebook_remove_page (notebook, page);
/* 必须刷新构件——
   这会迫使构件重绘自身 */
gtk_widget_draw(GTK_WIDGET(notebook), NULL);
}

void delete (GtkWidget *widget, GtkWidget *event, gpointer data)
{
    gtk_main_quit ();
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *table;
    GtkWidget *notebook;
    GtkWidget *frame;
    GtkWidget *label;
    GtkWidget *checkboxbutton;
    int i;
    char bufferf[32];
    char bufferl[32];

    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete), NULL);

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    table = gtk_table_new(3,6,FALSE);
    gtk_container_add (GTK_CONTAINER (window), table);

    /* 创建一个新的笔记本构件，将标签页放在顶部 */
    notebook = gtk_notebook_new ();
    gtk_notebook_set_tab_pos (GTK_NOTEBOOK (notebook), GTK_POS_TOP);
    gtk_table_attach_defaults(GTK_TABLE(table), notebook, 0,6,0,1);
    gtk_widget_show(notebook);

    /* 在笔记本构件后面追加几个页面 */
    for (i=0; i < 5; i++) {
        sprintf(bufferf, "Append Frame %d", i+1);
        sprintf(bufferl, "Page %d", i+1);

        frame = gtk_frame_new (bufferf);
        gtk_container_set_border_width (GTK_CONTAINER (frame), 10);
        gtk_widget_set_usize (frame, 100, 75);
        gtk_widget_show (frame);
    }
}
```

```
label = gtk_label_new (bufferf);
gtk_container_add (GTK_CONTAINER (frame), label);
gtk_widget_show (label);

label = gtk_label_new (bufferl);
gtk_notebook_append_page (GTK_NOTEBOOK (notebook), frame, label);
}

/* 在指定位置添加页面 */
checkboxbutton = gtk_check_button_new_with_label ("Check me please!");
gtk_widget_set_usize(checkboxbutton, 100, 75);
gtk_widget_show (checkboxbutton);

label = gtk_label_new ("Add page");
gtk_notebook_insert_page (GTK_NOTEBOOK (notebook), checkboxbutton, label, 2);

/* 向笔记本构件前插标签页 */
for (i=0; i < 5; i++) {
    sprintf(bufferf, "Prepend Frame %d", i+1);
    sprintf(bufferl, "PPage %d", i+1);

    frame = gtk_frame_new (bufferf);
    gtk_container_set_border_width (GTK_CONTAINER (frame), 10);
    gtk_widget_set_usize (frame, 100, 75);
    gtk_widget_show (frame);

    label = gtk_label_new (bufferf);
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_widget_show (label);

    label = gtk_label_new (bufferl);
    gtk_notebook_prepend_page (GTK_NOTEBOOK(notebook), frame, label);
}

/* 设置起始页 (第4页) */
gtk_notebook_set_page (GTK_NOTEBOOK(notebook), 3);

/* 创建一排按钮 */
button = gtk_button_new_with_label ("close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (delete), NULL);
gtk_table_attach_defaults(GTK_TABLE(table), button, 0,1,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("next page");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           (GtkSignalFunc) gtk_notebook_next_page,
                           GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 1,2,1,2);
gtk_widget_show(button);
```

```

button = gtk_button_new_with_label ("prev page");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           (GtkSignalFunc) gtk_notebook_prev_page,
                           GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 2,3,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("tab position");
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   (GtkSignalFunc) rotate_book,
                   GTK_OBJECT(notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 3,4,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("tabs/border on/off");
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   (GtkSignalFunc) tabsborder_book,
                   GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 4,5,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("remove page");
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   (GtkSignalFunc) remove_book,
                   GTK_OBJECT(notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 5,6,1,2);
gtk_widget_show(button);

gtk_widget_show(table);
gtk_widget_show(window);
gtk_main ();
return(0);
}
/* 示例结束 */

```

运行效果见图 10-7。请尝试一下添加、删除页面，切换页标签的位置，以及显示、隐藏页标签等功能。

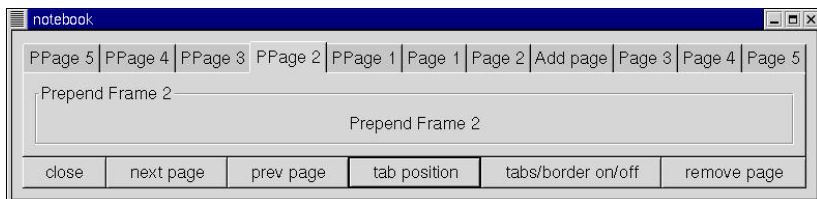


图10-7 笔记本构件

第11章 分栏列表构件GtkCList

GtkCList(分栏列表构件)是GtkList(列表构件)的替代品,但它提供更多的特性。分栏列表构件是多列列表构件,它有能力处理数千行的信息。每一列都可以有一个标题,而且可以是活动的。你还可以将函数绑定到列选择上。

11.1 创建分栏列表构件GtkCList

创建GtkCList构件的方法和创建其他构件的方法是类似的。有两种方法,一种是容易的,一种是难的。因为GtkCList可以有多列,因而在创建它之前,必须确定要创建的列表的列数。

```
GtkWidget *gtk_clist_new ( gint columns );  
GtkWidget *gtk_clist_new_with_titles( gint columns,  
                                       gchar *titles[] );
```

第一种方式很简单,而第二种需要作一些解释。每一列都可以有一个与之相联系的标题,标题可以是一个标签构件,或者是一个按钮,只要能够对我们的动作作出响应。如果要使用第二种方式,则必须提供一个指向标题文本的指针,指针数目应该与列数相等。当然,我们可以用第一种方式,然后再手工添加标题以达到相同的目的。

注意,分栏列表构件没有自己的滚动条,如果要提供滚动条功能,应该将分栏列表构件放在一个滚动窗口构件中。

11.2 操作模式

有几个可以用于改变分栏列表构件行为的属性。先看下面这个:

```
void gtk_clist_set_selection_mode( GtkCList *clist,  
                                   GtkSelectionMode mode );
```

就像函数名所暗示的一样,它设置了分栏列表的选择模式。第一个参数是要设置的分栏列表构件,第二个参数是单元的选择模式(取值在gtkenums.h中有定义)。目前,有下面这些模式可以使用:

- GTK_SELECTION_SINGLE: 选定内容为 NULL,或包含一个指向单个被选中项目的 GList 指针。
- GTK_SELECTION_BROWSE: 如果 GtkCList 中不包含构件,或只包含不敏感的构件,则选定内容为 NULL。否则,它包含一个指向 GList 结构的 GList 指针,因而包含一个列表项。
- GTK_SELECTION_MULTIPLE: 如果没有列表项被选中,选定内容为 NULL;或者选定内容是一个指向第一个被选中列表项的指针。然后依次指向 GList 结构中第二个被选中的列表项,等等。对 GtkCList 来说这是缺省模式。
- GTK_SELECTION_EXTENDED: 选中内容总是 NULL。

在 GTK 今后的版本中可能会增加其他模式。

还可以定义分栏列表构件的边框。使用以下函数完成定义:

```
void gtk_clist_set_shadow_type( GtkCList      *clist,
                               GtkShadowType border );
```

第二个参数可能的取值是：

GTK_SHADOW_NONE

GTK_SHADOW_IN

GTK_SHADOW_OUT

GTK_SHADOW_ETCHED_IN

GTK_SHADOW_ETCHED_OUT

11.3 操作分栏列表构件列标题

创建分栏列表构件时自动创建相应的标题按钮。标题一般处于分栏列表构件窗口的顶部，它可以是对鼠标点击响应的普通按钮，也可以仅仅是不会作任何响应的标签。有四个不同的函数调用帮助我们设置标题按钮的状态。

```
void gtk_clist_column_title_active( GtkCList *clist,
                                    gint      column );
void gtk_clist_column_title_passive( GtkCList *clist,
                                     gint      column );
void gtk_clist_column_titles_active( GtkCList *clist );
void gtk_clist_column_titles_passive( GtkCList *clist );
```

活动标题就是可以对用户动作响应的按钮标题，被动标题仅仅是一个标签。前两个函数激活或停用指定列的标题按钮，后两个激活或禁用整个分栏列表构件的按钮标题。

当然，有时候我们根本就不想使用标题按钮，那么可以用下面的函数显示或隐藏起来：

```
void gtk_clist_column_titles_show( GtkCList *clist );
void gtk_clist_column_titles_hide( GtkCList *clist );
```

有些情况下，标题对我们非常有用，需要有办法设置或改变它们。可以用以下函数来完成：

```
void gtk_clist_set_column_title( GtkCList *clist,
                                 gint      column,
                                 gchar     *title );
```

注意，一次只能设置一系列的标题。如果知道标题应该是什么，应该在开始时用前面介绍的gtk_clist_new_with_titles函数创建分栏列表并设置标题。这样可以节省编写代码的时间，并且让程序更小。当然也有一些情况下手工设置这些值可能更好。有时候不是所有的标题都是文本。GtkCList构件为我们提供的标题按钮实际上能够和所有的构件结合起来使用，例如，它可以和pixmap构件结合起来，在上面显示一幅图片。使用下面得用函数可以为标题按钮设置构件：

```
void gtk_clist_set_column_widget( GtkCList *clist,
                                   gint      column,
                                   GtkWidget *widget );
```

11.4 操纵列表

可以用以下的函数设置一系列的对齐方式：

```
void gtk_clist_set_column_justification( GtkCList      *clist,
```

```
gint          column,
GtkJustification justification );
```

GtkJustification参数类型可取以下值：

- GTK_JUSTIFY_LEFT：列中的文本左对齐。
- GTK_JUSTIFY_RIGHT：列中的文本右对齐。
- GTK_JUSTIFY_CENTER：列中的文本居中对齐。
- GTK_JUSTIFY_FILL：文本使用列中所有可用的空间。它通常会在单词间插入额外的空格(如果是一个单词，会在单个字母间加空格)。许多“所见即所得”的文本编辑器具有这种特性。

下面的函数非常重要，在设置 GtkCList构件时应该作为标准加以使用。创建构件时各列的宽度是依据它们的标题确定的，因为多数情况下这不一定正确，用下面的函数设置列宽度：

```
void gtk_clist_set_column_width( GtkCList *clist,
                                gint      column,
                                gint      width );
```

注意，宽度是以像素度量，而不是以字母度量的。这对某列的单元格的高度也是同样的，但是缺省值是当前字体的高度，这对应用程序来说并不是至关重要的。用下面的函数设置行高度：

```
void gtk_clist_set_row_height( GtkCList *clist,
                               gint      height );
```

再次强调，高度也是以像素度量的。

可以在没有用户干预的情况下在列表之间随意移动，不过，我们需要能够直接跳到所需的行和列，并且这个单元格应该是可见的。换句话说，就是我们应该能滚动列表，让单元格直接出现在可见的位置上。下面这个函数实现了这一点：

```
void gtk_clist_moveto( GtkCList *clist,
                      gint      row,
                      gint      column,
                      gfloat    row_align,
                      gfloat    col_align );
```

其中，row_align参数很重要。它是一个介于0.0和1.0之间的值，0.0意味着应该滚动列表，让所在行出现在顶部，如果row_align的值是1.0，该行会出现在底部。所有介于0.0到1.0之间的其他值表示位于顶部和底部之间的行。最后一个参数col_align和row_align的作用一样，不过0.0是指左边，1.0是指右边。

根据应用程序的需要，我们不需要滚动那些我们已经能够看见的单元格。所以，怎样才能知道单元格是不是可见的呢？有一个函数能够做到这一点：

```
GtkVisibility gtk_clist_row_is_visible( GtkCList *clist,
                                         gint      row );
```

返回值可能是以下几个值之一：

- GTK_VISIBILITY_NONE 不可见
- GTK_VISIBILITY_PARTIAL 部分可见
- GTK_VISIBILITY_FULL 全部可见

注意，现在只能知道某一行是否可见，还没有办法知道某列是否可见。但是你仍然可以获得部分信息，因为当调用上面的函数，返回值是 GTK_VISIBILITY_PARTIAL时，它的某一

部分是隐藏的，无法知道到底是被列表的顶部或底部挡住，还是这一行的某列在外边。

你还能改变特定列的前景色和背景色。这可用在当用户选中某行时标记该行。它有两个相关函数：

设置前景颜色：

```
void gtk_clist_set_foreground( GtkCList *clist,
                               gint        row,
                               GdkColor *color );
```

设置背景颜色：

```
void gtk_clist_set_background( GtkCList *clist,
                                gint        row,
                                GdkColor *color );
```

请注意颜色值必须是前面已经分配的。

11.5 向列表中添加行

可以用三种方法添加行。用下面函数可以在前面、后面加入行：

```
gint gtk_clist_prepend( GtkCList *clist,
                        gchar      *text[] );
gint gtk_clist_append( GtkCList *clist,
                        gchar      *text[] );
```

这两个函数返回整数值指明加入到列表中的行的索引号。可以用以下函数在指定位置插入一行：

```
void gtk_clist_insert( GtkCList *clist,
                       gint        row,
                       gchar      *text[] );
```

在这些函数中我们要提供一个指向要插入行的文本数组的指针。指针的个数应该等于列表的列数，如果 text[] 参数是 NULL，这一行的列中就没有文本。当我们想向列表中添加图片时可以这么做。

要注意，行和列的编号都是从 0 开始的。

用以下函数删除一行：

```
void gtk_clist_remove( GtkCList *clist,
                       gint        row );
```

还有一个函数可以用于删除列表中所有的行。这比对每一行调用一次 gtk_clist_remove 函数要来得快。

```
void gtk_clist_clear( GtkCList *clist );
```

还有两个很方便的函数可以用在当列表中要发生很大变化时。因为 GtkCList 在发生变化时要重绘自身，所以当列表中内容变化较大时，频繁重绘会让屏幕不停闪烁。最好的办法是先将列表“冻结”，然后更新列表，最后将其“解冻”。

“冻结”构件：

```
void gtk_clist_freeze( GtkCList *clist );
```

将构件“解冻”：

```
void gtk_clist_thaw( GtkCList *clist );
```

11.6 在单元格中设置文本和pixmap图片

单元格可以容纳pixmap图片、文本或同时包含两者。你可以使用以下函数：

```
void gtk_clist_set_text( GtkCList *clist,
                        gint row,
                        gint column,
                        const gchar *text );

void gtk_clist_set_pixmap( GtkCList *clist,
                          gint row,
                          gint column,
                          GdkPixmap *pixmap,
                          GdkBitmap *mask );

void gtk_clist_set_pixtext( GtkCList *clist,
                           gint row,
                           gint column,
                           gchar *text,
                           guint8 spacing,
                           GdkPixmap *pixmap,
                           GdkBitmap *mask );
```

这些函数应该很容易理解。它们都将要操作的 Clist构件作为第一个参数，第二、三个参数是行或列号，紧接着是要设置的数据。gtk_clist_set_pixtext 函数中的spacing参数是pixmap图片和文本起始点之间的间距。上面的函数中数据都被复制到 GtkCList构件中。

用以下函数可以读出相应的数据：

```
gint gtk_clist_get_text( GtkCList *clist,
                       gint row,
                       gint column,
                       gchar **text );

gint gtk_clist_get_pixmap( GtkCList *clist,
                          gint row,
                          gint column,
                          GdkPixmap **pixmap,
                          GdkBitmap **mask );

gint gtk_clist_get_pixtext( GtkCList *clist,
                           gint row,
                           gint column,
                           gchar **text,
                           guint8 *spacing,
                           GdkPixmap **pixmap,
                           GdkBitmap **mask );
```

返回的指针都是指向存储在构件内部的数据指针，所以不应该被修改或释放。引用的数据没有必要将不感兴趣的数据全部读出。任何返回值指针（除了 GtkCList构件）都可以是NULL。所以如果我们只想从类型为 pixtext的单元格中读出文本，应该像下面这样做，假定clist, row, column都已经存在：

```
gchar *mytext;
gtk_clist_get_pixtext(clist, row, column,
                     &mytext, NULL, NULL, NULL);
```

还有与上面内容相关的几个函数，下面这个函数将返回指定单元格的数据类型：

```
GtkCellType gtk_clist_get_cell_type( GtkCList *clist,
                                     gint         row,
                                     gint         column );
```

返回单元格内的数据类型，返回值可能是以下值之一：

GTK_CELL_EMPTY	单元格内是空的
GTK_CELL_TEXT	单元格内是文本
GTK_CELL_PIXMAP	单元格内是Pixmap图像
GTK_CELL_PIXTEXT	单元格内同时包含文本和图像
GTK_CELL_WIDGET	单元格内包含构件

还有一个函数可以用于设置单元格内水平和垂直方向上的缩排，缩排值是以像素度量的整数，它可以是正数也可以是负数。

```
void gtk_clist_set_shift( GtkCList *clist,
                          gint       row,
                          gint       column,
                          gint       vertical,
                          gint       horizontal );
```

11.7 存储数据指针

对GtkCList构件来说，可以为一行设置数据指针。指针对用户来说是不可见的。它可方便程序员将某一行与一些其他数据联系起来。

下面函数的意义从字面上就可以理解。

为指定的行设置数据：

```
void gtk_clist_set_row_data( GtkCList *clist,
                             gint       row,
                             gpointer    data );
```

与上面的函数类似，只是当销毁一行时调用由 destroy指定的回调函数：

```
void gtk_clist_set_row_data_full( GtkCList *clist,
                                  gint       row,
                                  gpointer    data,
                                  GtkDestroyNotify destroy );
```

获取指定行的数据：

```
gpointer gtk_clist_get_row_data( GtkCList *clist,
                                 gint       row );
```

在构件中搜索data所在的行。如果返回-1，则没有找到，或者没有匹配的：

```
gint gtk_clist_find_row_from_data( GtkCList *clist,
                                   gpointer    data );
```

11.8 处理选择

有几个函数可以让我们强行选中或取消一行的选择：

```
void gtk_clist_select_row( GtkCList *clist,
                           gint       row,
                           gint       column );
```



```
void gtk_clist_unselect_row( GtkCList *clist,
                             gint      row,
                             gint      column );
```

还有一个函数使用x、y坐标(例如,从鼠标指针得到坐标),返回坐标所在的行和列。

```
gint gtk_clist_get_selection_info( GtkCList *clist,
                                   gint      x,
                                   gint      y,
                                   gint      *row,
                                   gint      *column );
```

当我们监测到一些有趣的事件时(也许是鼠标指针的移动,或在列表的某处点击),可以读出鼠标的坐标值,找出鼠标正在列表的某一格。

11.9 信号

与其他构件一样, GtkCList有一些信号供我们使用。 GtkCList构件是从容器构件GtkContainer派生的,它有容器所具有的一些信号,还有下面这些附加信号:

- select_row: 选中一行时引发,该信号传递以下信息,依次是 GtkCList *clist、gint row、gint column、GtkEventButton *event。
- unselect_row: 用户对一行取消选择,引发这个信号。传递的信息与上一个信号一样。
- click_column: 选中某一列时引发。传递以下信息: GtkCList *clist、gint column。

所以,要将一个回调函数连接到 select_row信号上,回调函数应该像下面这样:

```
void select_row_callback(GtkWidget *widget,
                         gint row,
                         gint column,
                         GdkEventButton *event,
                         gpointer data);
```

回调函数用下面的形式连接到信号:

```
gtk_signal_connect(GTK_OBJECT( clist),
                  "select_row"
                  GTK_SIGNAL_FUNC(select_row_callback),
                  NULL);
```

11.10 GtkCList示例

```
/* CLiis示例开始 clist.c */
#include <gtk/gtk.h>
/* 用户点击"Add List按钮时的回调函数*/
void button_add_clicked( gpointer data )
{
    int indx;

    /* 字符串数组,用于加到list中,4行2列*/
    gchar *drink[4][2] = { { "Milk",      "3 Oz" },
                           { "Water",    "6 l" },
                           { "Carrots",  "2" },
                           { "Snakes",   "55" } };
```

/*下面将文本真正加到list中。对每行添加一次

```

    */
    for ( indx=0 ; indx < 4 ; indx++ )
        gtk_clist_append( (GtkCList *) data, drink[indx]);

    return;
}

/*用户点击"Clear List"按钮时的回调函数*/
void button_clear_clicked( gpointer data )
{
    /* 用gtk_clist_clear函数清除列表。比用
    *gtk_clist_remove函数逐行清除要快
    */
    gtk_clist_clear( (GtkCList *) data);

    return;
}

/* 用户点击"Hide/Show title"按钮时的回调函数*/
void button_hide_show_clicked( gpointer data )
{
    /* flag用于记录可见/不可见状态的标志。0 = 当前不可见 */
    static short int flag = 0;

    if (flag == 0)
    {
        /* 隐藏标题,将flag设置为1 */
        gtk_clist_column_titles_hide((GtkCList *) data);
        flag++;
    }
    else
    {
        /* 形式标题,将flag设置为0 */
        gtk_clist_column_titles_show((GtkCList *) data);
        flag--;
    }

    return;
}

/* 用户选中某一行时的回调函数 */
void selection_made( GtkWidget *clist,
                    gint row,
                    gint column,
                    GdkEventButton *event,
                    gpointer data )
{
    gchar *text;

    /* 取得存储在被选中的行和列的单元格上的文本
    * 当鼠标点击时,我们用text参数接收一个指针
    */

```

```

gtk_clist_get_text(GTK_CLIST(clist), row, column, &text);

/*打印一些关于选中了哪一行的信息*/
g_print("You selected row %d. More specifically you clicked in "
        "column %d, and the text in this cell is %s\n\n",
        row, column, text);

return;
}

int main( int      argc,
          gchar *argv[] )
{
    GtkWidget *window;
    GtkWidget *vbox, *hbox;
    GtkWidget *scrolled_window, *clist;
    GtkWidget *button_add, *button_clear, *button_hide_show;
    gchar *titles[2] = { "Ingredients", "Amount" };

    gtk_init(&argc, &argv);

    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_usize(GTK_WIDGET(window), 300, 150);

    gtk_window_set_title(GTK_WINDOW(window), "GtkCList Example");
    gtk_signal_connect(GTK_OBJECT(window),
                      "destroy",
                      GTK_SIGNAL_FUNC(gtk_main_quit),
                      NULL);

    vbox=gtk_vbox_new(FALSE, 5);
    gtk_container_set_border_width(GTK_CONTAINER(vbox), 5);
    gtk_container_add(GTK_CONTAINER(window), vbox);
    gtk_widget_show(vbox);

/* 创建一个滚动窗口构件，将GtkCList组装到里面。
 * 这样使得内容超出列表时，可以用滚动条浏览
 */
    scrolled_window = gtk_scrolled_window_new (NULL, NULL);
    gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                    GTK_POLICY_AUTOMATIC,
                                    GTK_POLICY_ALWAYS);

    gtk_box_pack_start(GTK_BOX(vbox), scrolled_window, TRUE, TRUE, 0);
    gtk_widget_show (scrolled_window);

/* 创建GtkCList构件。本例中，我们使用了两列 */
    clist = gtk_clist_new_with_titles( 2, titles);

/* 当作出选择时，我们要知道选择了哪一个单元格。使用
 * selection_made回调函数，代码在下面可以看见 */
    gtk_signal_connect(GTK_OBJECT(clist), "select_row",

```

```
GTK_SIGNAL_FUNC(selection_made),
NULL);

/* 不一定要设置边框的阴影,但是效果挺不错 */
gtk_clist_set_shadow_type (GTK_CLIST(clist), GTK_SHADOW_OUT);

/* 很重要的一点,我们设置列宽,让文本能容纳在列中。
 * 注意,列编号是从0开始的
 * 本例中是0和1
 */
gtk_clist_set_column_width (GTK_CLIST(clist), 0, 150);

/* 将GtkCList构件添加到滚动窗口构件中,然后显示 */
gtk_container_add(GTK_CONTAINER(scrolled_window), clist);
gtk_widget_show(clist);

/*创建按钮,把它们加到窗口中
 */
hbox = gtk_hbox_new(FALSE, 0);
gtk_box_pack_start(GTK_BOX(vbox), hbox, FALSE, TRUE, 0);
gtk_widget_show(hbox);

button_add = gtk_button_new_with_label("Add List");
button_clear = gtk_button_new_with_label("Clear List");
button_hide_show = gtk_button_new_with_label("Hide/Show titles");

gtk_box_pack_start(GTK_BOX(hbox), button_add, TRUE, TRUE, 0);
gtk_box_pack_start(GTK_BOX(hbox), button_clear, TRUE, TRUE, 0);
gtk_box_pack_start(GTK_BOX(hbox), button_hide_show, TRUE, TRUE, 0);

/* 为三个按钮的点击设置回调函数 */
gtk_signal_connect_object(GTK_OBJECT(button_add), "clicked",
                        GTK_SIGNAL_FUNC(button_add_clicked),
                        (gpointer) clist);
gtk_signal_connect_object(GTK_OBJECT(button_clear), "clicked",
                        GTK_SIGNAL_FUNC(button_clear_clicked),
                        (gpointer) clist);
gtk_signal_connect_object(GTK_OBJECT(button_hide_show), "clicked",
                        GTK_SIGNAL_FUNC(button_hide_show_clicked),
                        (gpointer) clist);

gtk_widget_show(button_add);
gtk_widget_show(button_clear);
gtk_widget_show(button_hide_show);

/* 界面已经完全设置好了,下面可以显示窗口,进入
 * gtk_main主循环
 */
gtk_widget_show(window);
gtk_main();
```

```
    return(0);  
}  
  
/* 示例结束 */
```

编译后，运行效果如图 11-1 所示。点击 Add List 按钮，向列表中添加列表项；点击 Clear List 按钮，清除所有列表项；点击 Hide/Show titles，轮换显示和隐藏列表标题。

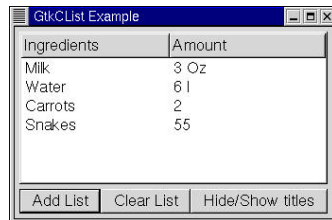


图11-1 GtkCList示例

第12章 树 构 件

树构件一般用于显示分层结构的数据。树构件本身是 `GtkTreeItem` 构件类型的垂直容器。树构件本身和分栏列表构件（`GtkCList`）差别并不大：它们都是由容器构件派生而来的，其中与容器相关的函数在树构件和分栏列表构件中同样起作用。差别在于树构件可以嵌套另外的树构件。下面简要介绍怎么使用树构件。

树构件有自己的X窗口，缺省情况下是白色的背景。还有，绝大多数树的函数与分栏列表的函数工作方法是一样的。然而，树构件并不是从分栏列表构件派生而来的，因此，这些函数不可以互换。

12.1 创建新树构件

用下面函数创建树构件：

```
GtkWidget *gtk_tree_new( void );
```

像分栏列表构件一样，当项目添加到树构件时，或当子树扩展时，树构件会自动扩展。因为这个原因，所以需要将树构件组装到一个滚动窗口构件中。滚动窗口构件的缺省尺寸是很小的，因此一般对滚动窗口要用 `gtk_widget_set_usize()` 函数设置它的缺省尺寸，以保证树构件足够大，可以看到其中的项目。

有了树构件后，就可以向其中添加项目了。现在，我们先介绍怎样创建树项构件：

```
GtkWidget *gtk_tree_item_new_with_label( gchar *label );
```

现在可以用下面的函数将树项构件添加到树构件中了：

```
void gtk_tree_append( GtkTree *tree,
                      GtkWidget *tree_item );
void gtk_tree_prepend( GtkTree *tree,
                      GtkWidget *tree_item );
```

注意，必须一次将全部树项添加到树构件中。没有与 `gtk_list_*_items()` 等价的函数。

12.1.1 添加一个子树

子树与树构件的创建方法类似。子树要加到一个树构件的树项下面，使用以下函数：

```
void gtk_tree_item_set_subtree( GtkTreeItem *tree_item,
                                GtkWidget *subtree );
```

在子树添加到一个树项之前或之后都不需要对子树调用 `gtk_widget_show()` 函数。然而，在调用 `gtk_tree_item_set_subtree()` 之前，必须将树项添加到一棵父树上。这是因为：从技术上来说，子树的父树并不是拥有它的树项构件（`GtkTreeItem`），而是树项所在的树。

当将子树添加到树项时，一个“+”或“-”符号会出现在旁边，用户可以点击它“展开”或“折叠”子树，也就是显示或隐藏子树。缺省状态树项是折叠起来的。注意，在折叠树项时，所有子树的选中状态仍然保持原有状态，这可能是用户不希望看到的。

12.1.2 处理选中的列表

与分栏列表构件 `GtkCList` 一样，树构件类型也有一个选择域。可以用下面的函数设置选择模式以控制树构件的行为：

```
void gtk_tree_set_selection_mode( GtkTree      *tree,
                                  GtkSelectionMode mode );
```

与各种选择模式相关的语义在分栏列表构件 `GtkCList` 中有详细的介绍。与 `GtkCList` 构件一样，当选择列表项或取消选择时，会引发 `select_child`、`unselect_child`、和 `selection_changed` 信号。然而，要使用这些信号，需要知道树构件怎样引发信号，以及如何查找选中的列表项。

所有的树构件都有自己的 X 窗口，它们能够接受事件，例如鼠标点击。然而，要让树构件的选择类型为 `GTK_SELECTION_SINGLE` 和 `GTK_SELECTION_BROWSE` 的树能够正常动作，选中项的列表必须是针对树构件层次上最顶层的（也就是所谓的“根”树）。

因而，除非你已经知道它是“根”树，否则，直接访问任意一个树构件的选择状态并不是什么好主意。应该使用 `GTK_TREE_SELECTION (Tree)` 宏，它将给出一个指向“根”树的选择列表的指针。当然，如果选择模式是 `GTK_SELECTION_MULTIPLE`（可以多选），这个选择列表也包含了其他不在所涉及到的子树下的树项。

最后，整个树都能引发 `select_child`（理论上还有 `unselect_child`）信号，但是，只有根树才能引发 `selection_changed` 信号。因而，如果想对一棵树和它的子树的 `select_child` 信号进行响应，必须对每一个子树调用 `gtk_signal_connect()` 函数。

12.1.3 树构件内部机制

Tree 的结构定义是这个样子：

```
struct _GtkTree
{
    GtkContainer container;
    GList *children;
    GtkTree* root_tree; /* owner of selection list */
    GtkWidget* tree_owner;
    GList *selection;
    guint level;
    guint indent_value;
    guint current_indent;
    guint selection_mode : 2;
    guint view_mode : 1;
    guint view_line : 1;
};
```

前面已经介绍过了直接访问树的选择状态的危险性。树构件定义中的其他部分也可以用宏或者函数访问。`GTK_IS_ROOT_TREE (Tree)` 返回一个布尔值，指明一个树是否是“根”树，而 `GTK_TREE_ROOT_TREE (Tree)` 宏返回指定树的“根”树（`GtkTree` 类型的值）（所以，要记住如果想使用 `gtk_widget_*`() 类型的函数，要用 `GTK_WIDGET (Tree)` 宏将对象转换为构件类型）。

与其直接访问 Tree 构件的子节点域，不如用 `GTK_CONTAINER (Tree)` 宏将树转换为一个

指针，然后将它传递到 `gtk_container_children()` 函数中。这样将为原来的列表创建一个副本。使用完后要用 `g_free()` 函数释放它，或用破坏性的方法遍历它。如下所示：

```
children = gtk_container_children (GTK_CONTAINER (tree));
while (children) {
    do_something_nice (GTK_TREE_ITEM (children->data));
    children = g_list_remove_link (children, children);
}
```

上面的 `tree_owner` 域只在子树中有定义，它指向容纳这个子树的树项构件；`level` 域指出特定树的嵌套层次，根部树的 `level` 是 0，每一个子树比它的双亲树的 `level` 大 1。这个域只有在树已在屏幕上绘出后才有设置值。

12.1.4 信号

```
void selection_changed( GtkTree *tree );
```

当树构件的选择区域发生变化时，会引发这个信号。也就是当选中树构件的子树或取消选择时。

```
void select_child( GtkTree *tree,
                  GtkWidget *child );
```

当选中树构件的子树时，将引发这个信号。当调用 `gtk_tree_select_item()` 和 `gtk_tree_select_child()`、或当鼠标按钮按下，调用 `gtk_tree_item_toggle()` 和 `gtk_item_toggle()` 函数时，也会引发该信号。当子树被添加到树上或者将子树从树上删除时会间接引发这个信号。

```
void unselect_child (GtkTree *tree,
                    GtkWidget *child);
```

当树的子树要被取消选中时，引发这个信号。

12.1.5 函数和宏

```
guint gtk_tree_get_type( void );
```

返回 `GtkTree` 构件的类型标识符

```
GtkWidget* gtk_tree_new( void );
```

创建新树，并返回指向 `GtkWidget` 对象类型的指针。如果创建失败，返回 `NULL`。

```
void gtk_tree_append( GtkTree *tree,
                     GtkWidget *tree_item );
```

将一个树项添加到树构件的后面。

```
void gtk_tree_prepend( GtkTree *tree,
                      GtkWidget *tree_item );
```

将一个树项添加到树构件的前面。

```
void gtk_tree_insert( GtkTree *tree,
                     GtkWidget *tree_item,
                     gint position );
```

在树构件中的指定位置插入树项。

```
void gtk_tree_remove_items( GtkTree *tree,
                           GList *items );
```

将一个树项列表(`GList *`形式的列表)从树构件中删除。注意，从一个树上删除一个树项会

解除它和它的子树，以及该子树的子树（如果有的话）的引用。如果只想删除一个树项，可以使用 `gtk_container_remove()` 函数。

```
void gtk_tree_clear_items( GtkTree *tree,
                           gint      start,
                           gint      end );
```

删除树构件中从 `start` 位置到 `end` 位置的树项。和上一个函数一样，它会解除树项及其子树的引用，因为这个函数只是构造一个列表，然后将列表传递给 `gtk_tree_remove_items()` 函数。

```
void gtk_tree_select_item( GtkTree *tree,
                           gint      item );
```

对指定 `item` 位置的树项引发 “ `select_item` ” 信号，并选中它（除非在信号处理函数中取消选择）。

```
void gtk_tree_unselect_item( GtkTree *tree,
                             gint      item );
```

对指定 `item` 位置的树项引发 “ `unselect_item` ” 信号，并取消选择。

```
void gtk_tree_select_child( GtkTree *tree,
                           GtkWidget *tree_item );
```

对子树项引发 “ `select_item` ” 信号，并且选中它。

```
void gtk_tree_unselect_child( GtkTree *tree,
                             GtkWidget *tree_item );
```

让子树项引发 “ `unselect_item` ” 信号，并取消选中状态。

```
gint gtk_tree_child_position( GtkTree *tree,
                             GtkWidget *child );
```

返回一个子构件在树中的位置。如果子构件不在树中，将返回 `-1`。

```
void gtk_tree_set_selection_mode( GtkTree *tree,
                                  GtkSelectionMode mode );
```

设置选择模式。模式可以是 `GTK_SELECTION_SINGLE`（缺省值），或 `GTK_SELECTION_BROWSE`、`GTK_SELECTION_MULTIPLE` 或 `GTK_SELECTION_EXTENDED`。它只对根部树有定义，也只有对根部树有意义，因为只有根部树才会被选择。它对子树设置没有任何效果，并简单忽略设置值。

```
void gtk_tree_set_view_mode( GtkTree *tree,
                             GtkTreeViewMode mode );
```

设置“视图模式”——可以是 `GTK_TREE_VIEW_LINE`（缺省值）或 `GTK_TREE_VIEW_ITEM`。视图模式从一个树构件传递到它的子树，并且不能对某个子树单独设置（这种说法也不完全正确）。

用术语解释“视图模式”相当暧昧，而在图中它决定当一个树的孩子被选中时，突出显示是什么样子。如果是 `GTK_TREE_VIEW_LINE`，整个树项会突出显示，如果是 `GTK_TREE_VIEW_ITEM`，只有子构件（通常是标签）突出显示。

```
void gtk_tree_set_view_lines( GtkTree *tree,
                              guint     flag );
```

是否在树项间划线。Flag 参数可以是 `TRUE`——代表在树项间划线，或者 `FALSE`，在树项间不划线。

```
GtkTree *GTK_TREE (gpointer obj);
```

将一个普通对象指针转换为 GtkTree * 指针。

```
GtkTreeClass *GTK_TREE_CLASS (gpointer class);
```

将一个普通类指针转换为 GtkTreeClass * 指针。

```
gint GTK_IS_TREE (gpointer obj);
```

判定一个普通指针是否指向一个 GtkTree 对象。

```
gint GTK_IS_ROOT_TREE (gpointer obj)
```

判定一个普通指针是否指向 GtkTree 构件以及它是否根树。虽然它可以接收任何指针，向函数传递一个并不指向树构件的指针也可能会引发问题。

```
GtkTree *GTK_TREE_ROOT_TREE (gpointer obj)
```

返回一个指向 GtkTree 构件的指针的根节点树。上面的警告同样适用。

```
GList *GTK_TREE_SELECTION( gpointer obj)
```

返回一个 GtkTree 构件的根树的选择列表。上面的警告同样适用。

12.2 树项构件 GtkTreeItem

树项构件和分栏列表项构件一样，是从 GtkItem 构件派生而来的，而 GtkItem 是从 GtkBin 构件派生而来的，因此，GtkTreeItem 是一种普通的容器，它只能包含一个子构件，且子构件可以是任何类型的。树项构件有一些额外的域，我们唯一需要关心的是 subtree——子树域。

TreeItem 结构的定义是下面这样的：

```
struct _GtkTreeItem
{
    GtkWidget item;
    GtkWidget *subtree;
    GtkWidget *pixmap_box;
    GtkWidget *plus_pix_widget, *minus_pix_widget;

    GList *pixmap;
    guint expanded : 1;
};
```

pixmap_box 域是一个事件盒构件，它捕捉对 “+” 或 “-” 的点击，控制树的展开或折叠。pixmap 域指向一个内部的数据结构。因为它总是可以使用 GTK_TREE_ITEM_SUBTREE (Item) 以一种相对安全的方式获得树项构件的 subtree 域，因而最好不要对树项构件的内部进行操作，除非你确实知道你在做什么。

因为它是直接从 GtkItem 构件派生而来的，所以它可以用 GTK_ITEM (TreeItem) 宏转换为一个指针。树项通常带一个标签，因此可以用 gtk_list_item_new_with_label() 方便地创建新的树项。同样的效果可以用下面的代码获得。这些代码实际上是逐字从 gtk_tree_item_new_with_label() 函数中复制而来：

```
tree_item = gtk_tree_item_new ();
label_widget = gtk_label_new (label);
gtk_misc_set_alignment (GTK_MISC (label_widget), 0.0, 0.5);

gtk_container_add (GTK_CONTAINER (tree_item), label_widget);
gtk_widget_show (label_widget);
```

因为上面的方法并不是强制性地 将标签构件添加到树项里面，所以可以将一个水平组装箱构件（GtkHBox）或箭头构件（GtkArrow）甚至一个笔记本构件（GtkNotebook）（虽然这样应用程序看起来会很奇怪）添加到树项中。

如果从一个子树中删除所有的树项，则这些树项会被销毁，并与子树解除父子关系，除非事前引用它。上层的树项会折叠起来，所以如果想要保留这些这些树项，应该做下面的工作：

```
gtk_widget_ref (tree);
owner = GTK_TREE(tree)->tree_owner;
gtk_container_remove (GTK_CONTAINER(tree), item);
if (tree->parent == NULL){
    gtk_tree_item_expand (GTK_TREE_ITEM(owner));
    gtk_tree_item_set_subtree (GTK_TREE_ITEM(owner), tree);
}
else
    gtk_widget_unref (tree);
```

可以对树项进行“拖放”，但要保证满足下面的条件：当调用 `gtk_widget_dnd_drag_set()` 或 `gtk_widget_dnd_drop_set()` 函数时要拖曳的树项和放下的树项都已经添加到一个树上，并且每一个父构件都有自己的父构件，一直到顶级窗口或对话框窗口为止。否则，会发生很奇怪的事。

12.2.1 信号

树项构件 `GtkTreeItem` 从 `GtkItem` 构件中继承了 `select`、`deselect` 和 `toggle` 信号。另外，它添加了两个自己的信号：`expand` 和 `collapse`。

```
void select( GtkItem *tree_item );
```

当一个 item 要被选中时引发这个信号，当用户点击它后也会引发，或当应用程序调用 `gtk_tree_item_select()`、`gtk_item_select()` 或 `gtk_tree_select_child()` 函数时。

```
void deselect( GtkItem *tree_item );
```

当取消选择树项时，或当用户在一个已选中的树项上点击时，会引发这个信号；当应用程序调用 `gtk_tree_item_deselect()` 或 `gtk_item_deselect()` 函数时也会引发该信号。对树项来说，在调用 `gtk_tree_unselect_child()` 和 `gtk_tree_select_child()` 函数时，也会引发该信号。

```
void toggle( GtkItem *tree_item );
```

当应用程序调用 `gtk_item_toggle()` 函数时，引发这个信号。效果是：如果该树项有一个父树的话，在树项上引发这个信号时，对树项的父树调用 `gtk_tree_select_child()`（绝不会调用 `gtk_tree_unselect_child()` 函数）；如果没有父树，高亮显示会反过来。

```
void expand( GtkTreeItem *tree_item );
```

当要展开一个树项的子树时，将引发整个信号。也就是当用户点击树项旁边的“+”时，或当应用程序调用 `gtk_tree_item_expand()` 函数时引发。

```
void collapse( GtkTreeItem *tree_item );
```

当一个树项的子树要折叠时，引发这个信号。也就是，当用户点击树项旁边的“-”，或当应用程序调用 `gtk_tree_item_collapse()` 函数时引发。

12.2.2 函数和宏

```
guint gtk_tree_item_get_type( void );
```

返回 “ GtkTreeItem ” 的类型标识符。

```
GtkWidget* gtk_tree_item_new( void );
```

创建新的树项构件GtkTreeItem。返回一个指向GtkWidget对象的指针。如果创建失败，将返回NULL。

```
GtkWidget* gtk_tree_item_new_with_label (gchar *label);
```

创建一个新的树项对象，该对象有一个唯一的子构件 GtkLabel。返回一个指向GtkWidget对象的指针。如果创建失败，则返回 NULL。

```
void gtk_tree_item_select( GtkTreeItem *tree_item );
```

这个函数是gtk_item_select (GTK_ITEM (tree_item))函数的一个封装，调用它会引发select信号。

```
void gtk_tree_item_deselect( GtkTreeItem *tree_item );
```

这个函数基本上是gtk_item_deselect (GTK_ITEM (tree_item))函数调用的一个封装。调用此函数时会引发deselect信号。

```
void gtk_tree_item_set_subtree( GtkTreeItem *tree_item,
                                GtkWidget *subtree );
```

这个函数将一个子树subtree添加到树项tree_item上。如果tree_item是展开的，会显示该子树；如果tree_item是折叠的，会隐藏该子树。还有，tree_item必须已经添加到树上。

```
void gtk_tree_item_remove_subtree( GtkTreeItem *tree_item );
```

删除树项的子树的所有树项（解除引用，然后销毁它，从子树向下一直到最末端的树项），然后删除该子树，并隐藏 “ + / - ” 符号。

```
void gtk_tree_item_expand( GtkTreeItem *tree_item );
```

使树项tree_item引发expand信号，展开该树项。

```
void gtk_tree_item_collapse( GtkTreeItem *tree_item );
```

使树项tree_item引发collapse信号，将该树项折叠。

```
GtkTreeItem *GTK_TREE_ITEM (gpointer obj)
```

将一个指针转换为GtkTreeItem*。

```
GtkTreeItemClass *GTK_TREE_ITEM_CLASS (gpointer obj)
```

将一个指针转换为GtkTreeItemClass类。

```
gint GTK_IS_TREE_ITEM (gpointer obj)
```

判定一个普通指针是否指向一个GtkTreeItem对象。

```
GtkWidget GTK_TREE_ITEM_SUBTREE (gpointer obj)
```

返回一个树项的子树(obj参数应该是指向一个GtkTreeItem对象的指针)。

12.3 树构件示例

下面的代码是树构件的一个示例。它在窗口中添加了一个树构件，并为所有相关对象的信号设置了回调函数。尝试一下，看看这些信号是怎样引发的，能用来做些什么。

```
/* 树构件示例开始tree.c */
```

```

#include <gtk/gtk.h>

/* for all the GtkItem:: and GtkTreeItem:: signals */
static void cb_itemsignal (GtkWidget *item, gchar *signame)
{
    gchar *name;
    GtkWidget *label;

    /* 它是从GtkBin派生而来，所以它只能有一个子构件 */
    label = GTK_LABEL (GTK_BIN (item)->child);
    /* 取得标签的文本 */
    gtk_label_get (label, &name);
    /* 获取树项所在树的层次 */
    g_print ("%s called for item %s->p, level %d\n", signame, name,
              item, GTK_TREE (item->parent)->level);
}

/* 注意，这个函数没有被调用过 */
static void cb_unselect_child (GtkWidget *root_tree, GtkWidget *child,
                               GtkWidget *subtree)
{
    g_print ("unselect_child called for root tree %p, subtree %p, child %p\n",
              root_tree, subtree, child);
}

/* 注意这个函数在用户点击树项时调用，不管它是否已经被选中 */
static void cb_select_child (GtkWidget *root_tree, GtkWidget *child,
                             GtkWidget *subtree)
{
    g_print ("select_child called for root tree %p, subtree %p, child %p\n",
              root_tree, subtree, child);
}

static void cb_selection_changed (GtkWidget *tree)
{
    GList *i;
    g_print ("selection_change called for tree %p\n", tree);
    g_print ("selected objects are:\n");

    i = GTK_TREE_SELECTION(tree);
    while (i){
        gchar *name;
        GtkWidget *label;
        GtkWidget *item;

        /* 从列表节点获得一个指向GtkWidget类型的指针 */
        item = GTK_WIDGET (i->data);
        label = GTK_LABEL (GTK_BIN (item)->child);
        gtk_label_get (label, &name);
        g_print ("\t%s on level %d\n", name, GTK_TREE
                  (item->parent)->level);
    }
}

```

```
    i = i->next;
}
}

int main (int argc, char *argv[])
{
    GtkWidget *window, *scrolled_win, *tree;
    static gchar *itemnames[] = {"Foo", "Bar", "Baz", "Quux",
                                "Maurice"};

    gint i;

    gtk_init (&argc, &argv);

    /* a generic toplevel window */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_signal_connect (GTK_OBJECT(window), "delete_event",
                        GTK_SIGNAL_FUNC (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER(window), 5);

    /* 创建一个滚动窗口 */
    scrolled_win = gtk_scrolled_window_new (NULL, NULL);
    gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_win),
                                   GTK_POLICY_AUTOMATIC,
                                   GTK_POLICY_AUTOMATIC);
    gtk_widget_set_usize (scrolled_win, 150, 200);
    gtk_container_add (GTK_CONTAINER(window), scrolled_win);
    gtk_widget_show (scrolled_win);

    /* 创建一个根树 */
    tree = gtk_tree_new();
    g_print ("root tree is %p\n", tree);
    /* connect all GtkTree:: signals */
    gtk_signal_connect (GTK_OBJECT(tree), "select_child",
                        GTK_SIGNAL_FUNC(cb_select_child), tree);
    gtk_signal_connect (GTK_OBJECT(tree), "unselect_child",
                        GTK_SIGNAL_FUNC(cb_unselect_child), tree);
    gtk_signal_connect (GTK_OBJECT(tree), "selection_changed",
                        GTK_SIGNAL_FUNC(cb_selection_changed), tree);
    /* 将这个树加到滚动窗口中 */
    gtk_scrolled_window_add_with_viewport (GTK_SCROLLED_WINDOW(scrolled_win),
                                           tree);

    /* 设置选择模式 */
    gtk_tree_set_selection_mode (GTK_TREE(tree),
                                GTK_SELECTION_MULTIPLE);

    /* 显示树构件 */
    gtk_widget_show (tree);

    for (i = 0; i < 5; i++){
        GtkWidget *subtree, *item;
        gint j;
```



```
/* 创建一个树项 */
item = gtk_tree_item_new_with_label (itemnames[i]);
/* 为所有的树项信号设置回调函数 */
gtk_signal_connect (GTK_OBJECT(item), "select",
                    GTK_SIGNAL_FUNC(cb_itemsignal), "select");
gtk_signal_connect (GTK_OBJECT(item), "deselect",
                    GTK_SIGNAL_FUNC(cb_itemsignal), "deselect");
gtk_signal_connect (GTK_OBJECT(item), "toggle",
                    GTK_SIGNAL_FUNC(cb_itemsignal), "toggle");
gtk_signal_connect (GTK_OBJECT(item), "expand",
                    GTK_SIGNAL_FUNC(cb_itemsignal), "expand");
gtk_signal_connect (GTK_OBJECT(item), "collapse",
                    GTK_SIGNAL_FUNC(cb_itemsignal), "collapse");
/* 将树项添加到树上 */
gtk_tree_append (GTK_TREE(tree), item);
/* Show it - this can be done at any time */
gtk_widget_show (item);
/* 为树项创建子树 */
subtree = gtk_tree_new();
g_print ("-> item %s->%p, subtree %p\n", itemnames[i], item,
          subtree);

/* 如果想要这些信号对子树的子构件也起作用，下面的代码就是必要的。
 * 注意，"selection_change"信号总会在根树上引发 */
gtk_signal_connect (GTK_OBJECT(subtree), "select_child",
                    GTK_SIGNAL_FUNC(cb_select_child), subtree);
gtk_signal_connect (GTK_OBJECT(subtree), "unselect_child",
                    GTK_SIGNAL_FUNC(cb_unselect_child), subtree);
/* 下面一句代码没有效果，因为对子树来说，它已经被全部忽略了。 */
gtk_tree_set_selection_mode (GTK_TREE(subtree),
                             GTK_SELECTION_SINGLE);
/* 下面的代码也没有作用，不过理由与上面不一样：
 * 树构件的"view_mode"和"view_line"值是从根
 * 树开始从上往下传递的，所以，在后面设置有可能
 * 会产生不可预料的结果 */
gtk_tree_set_view_mode (GTK_TREE(subtree), GTK_TREE_VIEW_ITEM);
/* 设置树项的子树。注意，只有将树项加到父树上之后才能做这件事 */
gtk_tree_item_set_subtree (GTK_TREE_ITEM(item), subtree);

for (j = 0; j < 5; j++){
    GtkWidget *subitem;

    /* 创建一个子树 */
    subitem = gtk_tree_item_new_with_label (itemnames[j]);
    /* 连接所有树项的回调函数 */
    gtk_signal_connect (GTK_OBJECT(subitem), "select",
                        GTK_SIGNAL_FUNC(cb_itemsignal), "select");
    gtk_signal_connect (GTK_OBJECT(subitem), "deselect",
                        GTK_SIGNAL_FUNC(cb_itemsignal), "deselect");
    gtk_signal_connect (GTK_OBJECT(subitem), "toggle",
```

```
        GTK_SIGNAL_FUNC(cb_itemsignal), "toggle");
gtk_signal_connect (GTK_OBJECT(subitem), "expand",
        GTK_SIGNAL_FUNC(cb_itemsignal), "expand");
gtk_signal_connect (GTK_OBJECT(subitem), "collapse",
        GTK_SIGNAL_FUNC(cb_itemsignal), "collapse");
g_print ("-> -> item %s->%p\n", itemnames[j], subitem);
/* 将它加到父树构件上 */
gtk_tree_append (GTK_TREE(subtree), subitem);
/* 显示这个构件 */
gtk_widget_show (subitem);
    }
}

/* 显示窗口，然后进入主循环 */
gtk_widget_show (window);
gtk_main();
return 0;
}

/* 示例结束 */
```

此示例的运行结构如图 12-1 所示。

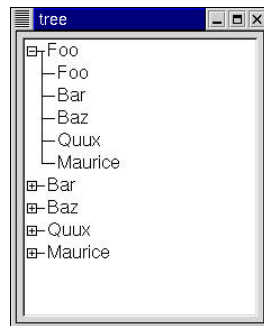


图12-1 树构件示例

第13章 GnomeApp构件和GnomeUIInfo

13.1 主窗口GnomeApp

所有的Gnome应用程序，除极少数有特殊需要的以外，都可以用GnomeApp作为其主窗口。GnomeApp构件是GtkWindow的子类，它在基本的顶级窗口上增加了很方便的菜单和工具条处理能力，如图13-1所示。用户可以配置GnomeApp构件的下列特性：

- 菜单和工具条可以与窗口分离开，或者在窗口上重新安排位置。
- 用户可以选择禁止菜单和工具条与Gnome应用程序的窗口分离。
- 用户可以选择是否在应用程序的菜单上显示小图标。

今后，GnomeApp构件还会添加更多的特性。

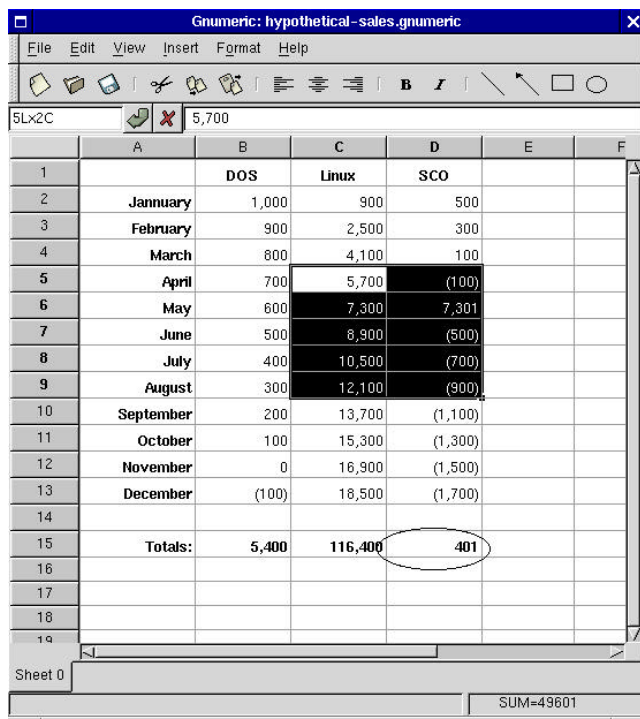


图13-1 Gnumeric电子表格软件，用GnomeApp构件创建界面

GnomeApp有一个与其他构件类似的创建函数，见下面的函数列表。第一个参数 app_id 是一个Gnome与应用程序打交道的内部名称。它应该与传递到 gnome_init() 函数中的 app_id 完全一样，一般来说，可以使用应用程序可执行文件的名称。第二个参数是应用程序窗口的标题，如果设为 NULL，则不为窗口设置标题。

```
#include <libgnomeui/gnome-app.h>
GtkWidget* gnome_app_new(gchar* app_id,
```

```
gchar* title)
```

GnomeApp构件有一个唯一的“内容区”，你可以将应用程序的主要功能放在该区域。在中心区域的四边，可以放置工具条、菜单条、状态条等。函数列表列出了相关的函数。

这些函数很容易理解，一般从字面上就可以知道其作用。它们的主要作用是在 GnomeApp 构件的合适位置放置所需要的构件。为 GnomeApp 创建菜单条、工具条、状态条也有很简单的方法。

添加构件到GnomeApp上，可输入以下语句：

```
#include <libgnomeui/gnome-app.h>
/* 将构件contents添加到GnomeApp构件app的窗口上 */
void gnome_app_set_contents(GnomeApp* app,
                             GtkWidget* contents)

/* 将菜单条menubar添加到GnomeApp构件app的窗口上 */
void gnome_app_set_menus(GnomeApp* app,
                          GtkMenuBar* menubar)

/* 将工具条toolbar添加到GnomeApp构件app上 */
void gnome_app_set_toolbar(GnomeApp* app,
                           GtkToolBar* toolbar)

/* 将状态条statusbar添加到GnomeApp构件上 */
void gnome_app_set_statusbar(GnomeApp* app,
                              GtkWidget* statusbar)
```

13.2 GnomeUIInfo

13.2.1 创建GnomeUIInfo

Gtk提供了两种方法来为应用程序创建菜单。但是，用这两种方法创建一个很大的菜单是很冗长乏味的，特别是如果菜单还带有图标和快捷键时更是如此。Gnome提供了一个简单的解决方案。为每一个菜单项创建一个 GnomeUIInfo 模板，并列出它的一些特性：名称、图标、快捷键等。Gnome库函数会自动地用 GnomeUIInfo 数组模板创建菜单。你也可以用同样的方法创建工具条。

下面是GnomeUIInfo. 结构的声明：

```
typedef struct {
    GnomeUIInfoType type;
    gchar* label;
    gchar* hint;
    gpointer moreinfo;
    gpointer user_data;
    gpointer unused_data;
    GnomeUIPixmapType pixmap_type;
    gpointer pixmap_info;
    guint accelerator_key;
    GdkModifierType ac_mods;
    GtkWidget* widget;
} GnomeUIInfo;
```

填充上面的结构最方便的方法就是用一段静态的初始化程序（当然，如果愿意，也可以动态地创建）。Gnome函数可以接受一个 GnomeUIInfo数组，同时，还有一些宏可以简化、标准化最常用的静态初始化程序。下面是一个典型的例子——一个“文件”菜单：

```
static GnomeUIInfo file_menu[] = {
    GNOMEUIINFO_MENU_NEW_ITEM(N_("New Window"),
                                N_("Create a new text viewer window"),
                                new_app_cb, NULL),

    /*"打开" (Open) 菜单项*/
    GNOMEUIINFO_MENU_OPEN_ITEM(open_cb, NULL),

    /*"另存为" (Save As) 菜单项*/
    GNOMEUIINFO_MENU_SAVE_AS_ITEM(save_as_cb, NULL),

    /*分隔线*/
    GNOMEUIINFO_SEPARATOR,

    /*"关闭" (Close) 菜单项*/
    GNOMEUIINFO_MENU_CLOSE_ITEM(close_cb, NULL),

    /*"退出" (Exit) 菜单项*/
    GNOMEUIINFO_MENU_EXIT_ITEM(exit_cb, NULL),

    /*菜单结束*/
    GNOMEUIINFO_END
};
```

然而，大多数情况下，是不能用上面的宏来创建菜单的，所以有时候还得自己动手指定结构的每一个成员：

```
{
    GNOME_APP_UI_ITEM, N_("Select All"),
    N_("Select all cells in the spreadsheet"),
    select_all_cb, NULL,
    NULL, 0, 0, 'a', GDK_CONTROL_MASK
}
```

下面我们简要介绍 GnomeUIInfo结构中各个成员的含义：

- type 是一个 GnomeUIInfoType 枚举类型的值，也是类型标记，参看表 13-1。
- label 是菜单或工具条按钮上的文本。一般情况下，它应该用 N_() 宏作国际化标注。
- hint 是菜单项或按钮功能的描述，对工具条来说，它以工具提示文本的形式显示。对菜单条来说，它可以显示在状态条上。
- moreinfo 依赖于数据项的类型，参见表 13-1。
- user_data 如果这个菜单项有回调函数，user_data 会被传递到回调函数。
- unused_data 应该设置为 NULL，现在还没有用到。Gnome 的后续版本也许会用到它。
- pixmap_type 是一个 GnomeUIPixmapType 枚举类型的值，它的作用是指定它的下一个成员 pixmap_info 的类型。
- pixmap_info 可以是原始的 pixmap 数据、一个文件名，或者是一个 Gnome 内置 pixmap 图

片。

- `accelerator_key` 是这个菜单项的快捷键。可以用一个字符，比如 “ a ”，或者是 `gdk/gdkkeysyms.h`中定义的一个值来表示。
- `ac_mods` 是一个用于快捷键的组合键屏蔽值。
- `widget` 应该设置为 `NULL`。当Gnome创建菜单项或工具条按钮时，会将构件填充在其中。如果想要用某种方式操纵该构件，可以检索到它。

菜单项的名字的下划线用于标志菜单项的快捷键。翻译程序会根据需要将下划线移开，让菜单文本在其他国家的语言中是可读的。Gnome会分析菜单项名称，并取得加速键，然后将下划线删除。

表中概括了GnomeUIInfo结构中type域的可能取值。其中有几个最可能的取值，但是其他几个值由Gnome内部使用。下表对应用程序来说应该足够了。

表13-1 GnomeUIInfoType值

GnomeUIInfoType	描 述	moreinfo域
<code>GNOME_APP_UI_ENDOFINFO</code>	终止一个GnomeUIInfo表	None
<code>GNOME_APP_UI_ITEM</code>	普通菜单或工具条项(或在 无线按钮组中的单选项)	回调函数
<code>GNOME_APP_UI_TOGGLEITEM</code>	切换/检查项	回调函数
<code>GNOME_APP_UI_RADIOITEMS</code>	无线按钮项组	组中的无线按钮项数组
<code>GNOME_APP_UI_SUBTREE</code>	子菜单	菜单子树中的GnomeUIInfo数组
<code>GNOME_APP_UI_SEPARATOR</code>	分隔线	None
<code>GNOME_APP_UI_HELP</code>	帮助项	要加载的帮助节点

要创建一个完整的菜单树，可以用 `GNOMEUIINFO_SUBTREE()`宏生成一个指向上级菜单表的指针（注意，这里的 `file_menu`就是前面创建初始化的GnomeUIInfo结构）：

```
static GnomeUIInfo main_menu[] = {
    GNOMEUIINFO_SUBTREE(N_("File"), file_menu),
    GNOMEUIINFO_END
};
```

不过，在这个特殊情况中，还有一个更好的宏：

```
static GnomeUIInfo main_menu[] = {
    GNOMEUIINFO_MENU_FILE_TREE(file_menu),
    GNOMEUIINFO_END
};
```

这个宏的主要优点是标准化。它保证所有的Gnome应用程序的文件菜单有同样的名称和快捷键。还有一些类似的宏，请参见 `libgnomeui/gnome-app-helper.h`头文件

13.2.2 将GnomeUIInfo转换为构件

一旦有了菜单表，Gnome就会对它进行处理，并将它转换成构件。可以使用下表中的函数来完成转换。

函数列表：由GnomeUIInfo创建构件

```
#include <libgnomeui/gnome-app-helper.h>
/*将uuinfo转换为菜单并连接到GnomeApp窗口app上*/
```



```

void gnome_app_create_menus(GnomeApp* app,
                             GnomeUIInfo* uiinfo)

/*与上一个函数差不多,但是user_data会覆盖uiinfo中的user_data*/
void gnome_app_create_menus_with_data(GnomeApp* app,
                                       GnomeUIInfo* uiinfo,
                                       gpointer user_data)

/*将uiinfo转换为工具条,然后添加到GnomeApp窗口app上*/
void gnome_app_create_toolbar(GnomeApp* app,
                              GnomeUIInfo* uiinfo)

/*与上一个函数差不多,但是user_data会覆盖uiinfo中的user_data*/
void gnome_app_create_toolbar_with_data(GnomeApp* app,
                                         GnomeUIInfo* uiinfo,
                                         gpointer user_data)

/*将uiinfo转换为工具条构件。可以将这生成的工具条添加到普通的 GtkWidget上*/
void gnome_app_fill_toolbar(GtkToolbar* toolbar,
                           GnomeUIInfo* uiinfo,
                           GtkAccelGroup* accel_group)

/*与上一个函数类似,但是data会覆盖uiinfo中的userdata*/
void gnome_app_fill_toolbar_with_data(GtkToolbar* toolbar,
                                       GnomeUIInfo* uiinfo,
                                       GtkAccelGroup* accel_group,
                                       gpointer data)

/*用uiinfo创建一个GtkMenuShell,可以将这个菜单项添加到GtkMenu上*/
void gnome_app_fill_menu(GtkMenuShell* menushell,
                        GnomeUIInfo* uiinfo,
                        GtkAccelGroup* accel_group,
                        gboolean uline_accels,
                        gint pos)

/*与上一个函数类似,但是user_data会覆盖uiinfo中的userdata*/
void gnome_app_fill_menu_with_data(GtkMenuShell* menushell,
                                    GnomeUIInfo* uiinfo,
                                    GtkAccelGroup* accel_group,
                                    gboolean uline_accels,
                                    gint pos,
                                    gpointer user_data)

```

如果用GnomeApp作为主窗口, gnome_app_create_menus() 和gnome_app_create_toolbar() 函数用所提供的 GnomeUIInfo表创建菜单条和工具条,然后将它们连接到 GnomeApp窗口上。多数情况下,用这些函数就可以了。所有工作,如设置菜单和工具条等,都是由 Gnome自动完成的。上面的每个函数都有一个以 _with_data()结尾的变体,变体函数中的 user_data参数都会覆盖GnomeUIInfo结构中的user_data参数。

如果有更特殊的需要,可以手工填充一个菜单条或者工具条,然后把它添加到容器中。

上面最后四个函数就是用于完成这个工作的。填充函数需要指定一个快捷键组，对 GnomeApp 来说，在构件结构中已经有了一个快捷键组（`accel_group`成员）。菜单的填充函数带两个参数 `accel_group`和`uline_accels`可以切换是否分析 GnomeUIInfo结构标签中的下划线以提取快捷键，`uline_accels`为TRUE时提取快捷键放到 `accel_group`中，否则，不提取快捷键。对填充菜单，`pos`参数指定应该在 GtkMenuShell 中的什么位置开始插入菜单项。

当用GnomeUIInfo 表创建菜单条和工具条时，指向单个菜单项或工具条按钮构件的指针存放在每个 GnomeUIInfo 结构的`widget`成员中。可以用这些指针访问单个构件，例如，如果创建了一个检查菜单项，也许想设置检查项的状态。如果想手工创建部分菜单，这个指针也是很有用的，例如，可以创建一个空的菜单子树项，然后手工创建子树的内容。

第14章 状态条构件

14.1 状态条构件简介

Gtk+ 构件库中有一个状态条构件 `GtkStatusbar`，Gnome 构件库中也有一个状态条 `GnomeAppBar`。这两者之间没有多大差别，所以选择哪一个构件并没有什么特别的关系。状态条一般用来显示一些提示性的信息。因为有的用户，特别是新用户可能根本就注意不到状态条上的信息，因此，不能将在状态条上显示信息（特别是重要信息）作为唯一的提示方式。

为GnomeApp构件添加状态条很简单。只需调用 `gnome_app_set_statusbar` 函数，并将第二个参数设置为已经创建好的 `statusbar` 构件。当鼠标指向某个菜单时，可以用状态条显示菜单的帮助。Gnome有几个很方便的函数可以实现这种功能。

函数列表：设置状态条

```
#include <libgnomeui/gnome-app.h>
void
gnome_app_set_statusbar(GnomeApp* app,
                        GtkWidget* statusbar)
```

14.2 GnomeAppBar构件

并没有特别的理由选择是用 `GnomeAppBar` 还是 `GtkStatusbar` 作为状态条，主要区别在于它们拥有不同的API函数。`GnomeAppBar` 构件是后写的，目的在于以下几点：

- 简化 `GtkStatusbar` 构件的API调用。
- 支持Netscape风格的状态条，在状态条上显示一个进度条。
- 最终目的是要支持像Emacs编辑器的“minibuffer”功能那样的交互功能。不过，这个功能在Gnome 1.0中还没有实现。

用 `gnome_appbar_new()` 函数能够创建 `GnomeAppBar` 构件。用这个构造函数还可以配置 `GnomeAppBar` 构件的功能：有或者没有进度条，有或者没有状态文本区，可以或不可以与用户交互。注意，必须有一个进度条或状态文本区。其中，`GnomePreferencesType` 是一种扩展型的布尔值：

- `GNOME_PREFERENCES_NEVER` 表明 `GnomeAppBar` 构件是不可交互的。
- `GNOME_PREFERENCES_USER` 表明如果用户已经在Gnome环境设置中激活这种特性，`GnomeAppBar` 就是交互的。
- `GNOME_PREFERENCES_ALWAYS` 表明 `GnomeAppBar` 总是可交互的。

Gnome 1.0还没有完全实现交互性，所以要避免使用 `GNOME_PREFERENCES_ALWAYS`。还有一些实验性的Gnome函数，可以用于提取某些用户交互动作，并允许用户在对话框和Emacs风格的“minibuffer”之间作出选择。当Gnome得到进一步发展后，`GNOME_PREFERENCES_USER` 会起作用，即使并没有明确使用“交互性”。建议将 `GnomePreferencesType` 设置为 `GNOME_PREFERENCES_USER`。

函数列表：创建GnomeAppBar构件

```
#include <libgnomeui/gnome-appbar.h>
GtkWidget*
gnome_appbar_new(gboolean has_progress,
                 gboolean has_status,
                 GnomePreferencesType interactivity)
```

GnomeAppBar的用法很简单。进度条元素代表一个 GtkProgress接口，要使用该接口，只需用 gnome_appbar_get_progress()函数将 GtkProgress 提取出来，然后就可以使用与 GtkProgress构件的相关函数对它进行操作了。注意，不要假想 Progress Bar接口是 GtkProgress 的子类；不要将它转换为 GtkProgressBar类型的指针。

函数列表：提取GtkProgress

```
#include <libgnomeui/gnome-appbar.h>
GtkProgress* gnome_appbar_get_progress(GnomeAppBar* appbar)
```

状态文本存储在一个栈中。当 GnomeAppBar 刷新时，显示栈中最上面的元素。每次对栈进行操作时，GnomeAppBar 都会刷新。所以将状态文本压入栈时，该文本就会显示出来。

状态文本还有另外两种设置方法。你可以设置一些“缺省”文本，如果栈是空的，会显示缺省文本。缺省的“缺省”文本是空字符串。你还可以仅设置状态文本而不改变栈，则“暂时”文本立即显示在状态文本区，但不存储在栈中。在下次刷新时（下次压入、弹出或设置缺省文本时），该文本会永久消失，并被栈顶的值所取代。

下面的函数列表列出了操纵状态文本的函数。 gnome_appbar_set_status()函数用于设置“暂时”状态文本； gnome_appbar_refresh()强行刷新而不改变栈——这样可以保证“暂时”文本已经清除。其他的函数意义都很明显。

注意 可以将GnomeAppBar用作简单的标签，它一次显示一条信息，且总是取代前一条信息，只要设置缺省文本或暂时文本就可以了，根本不需使用栈。

函数列表：设置GnomeAppBar的文本

```
#include <libgnomeui/gnome-appbar.h>
void gnome_appbar_set_status(GnomeAppBar* appbar,
                             const gchar* status)
void gnome_appbar_set_default(GnomeAppBar* appbar,
                             const gchar* default_status)
void gnome_appbar_push(GnomeAppBar* appbar,
                      const gchar* status)
void gnome_appbar_pop(GnomeAppBar* appbar)
void gnome_appbar_clear_stack(GnomeAppBar* appbar)
void gnome_appbar_refresh(GnomeAppBar* appbar)
```

14.3 状态条构件GtkStatusbar

GtkStatusbar是一个简单的构件，一般用来显示文本消息。它将文本消息压入到一个栈里面，当弹出当前消息时，将重新显示前一条文本消息。

为了让应用程序的不同部分使用同一个状态条显示消息，状态条构件使用上下文标识符来识别不同“用户”。在栈顶部的消息就是要显示的消息，不管它的上下文是什么。消息在栈里面是以先进后出的方式保存的，而不是按上下文标识符顺序。

状态条构件用下面的函数创建：

```
GtkWidget *gtk_statusbar_new( void );
```

用一个上下文的简短文本描述调用下面的函数，可以获得新的上下文标识符：

```
guint gtk_statusbar_get_context_id( GtkWidget *statusbar,
                                   const gchar *context_description );
```

有三个函数用来操作状态条：

```
guint gtk_statusbar_push( GtkWidget *statusbar,
                          guint context_id,
                          gchar *text );
void gtk_statusbar_pop( GtkWidget *statusbar,
                       guint context_id );
void gtk_statusbar_remove( GtkWidget *statusbar,
                           guint context_id,
                           guint message_id );
```

第一个函数gtk_statusbar_push用于将新消息加到状态栏中，它返回消息的上下文标识符。这个标识符可以用在 gtk_statusbar_remove函数中将该消息从状态条的栈中删除。函数gtk_statusbar_pop删除在栈中给定上下文标识符的最上面的一条消息。

下面的例子创建了一个状态条和两个按钮，一个将消息压入到状态条栈中，另一个将最上面一条消息弹出。

```
/* 状态条示例开始 statusbar.c */
#include <gtk/gtk.h>
#include <glib.h>

GtkWidget *status_bar;
void push_item (GtkWidget *widget, gpointer data)
{
    static int count = 1;
    char buff[20];
    g_snprintf(buff, 20, "Item %d", count++);
    gtk_statusbar_push( GTK_STATUSBAR(status_bar),
                       GPOINTER_TO_INT(data), buff);
    return;
}

void pop_item (GtkWidget *widget, gpointer data)
{
    gtk_statusbar_pop( GTK_STATUSBAR(status_bar),
                      GPOINTER_TO_INT(data) );
    return;
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *button;
    gint context_id;
```

```

gtk_init (&argc, &argv);

/* 创建新窗口 */
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_widget_set_usize( GTK_WIDGET (window), 200, 100);
gtk_window_set_title(GTK_WINDOW (window), "GTK Statusbar Example");
gtk_signal_connect(GTK_OBJECT (window), "delete_event",
                  (GtkSignalFunc) gtk_exit, NULL);

vbox = gtk_vbox_new(FALSE, 1);
gtk_container_add(GTK_CONTAINER(window), vbox);
gtk_widget_show(vbox);
status_bar = gtk_statusbar_new();
gtk_box_pack_start (GTK_BOX (vbox), status_bar, TRUE, TRUE, 0);
gtk_widget_show (status_bar);

context_id = gtk_statusbar_get_context_id(
                  GTK_STATUSBAR(status_bar),
                  "Statusbar example");

button = gtk_button_new_with_label("push item");
gtk_signal_connect(GTK_OBJECT(button), "clicked",
                  GTK_SIGNAL_FUNC (push_item), GINT_TO_POINTER(context_id) );
gtk_box_pack_start(GTK_BOX(vbox), button, TRUE, TRUE, 2);
gtk_widget_show(button);

button = gtk_button_new_with_label("pop last item");
gtk_signal_connect(GTK_OBJECT(button), "clicked",
                  GTK_SIGNAL_FUNC (pop_item), GINT_TO_POINTER(context_id) );
gtk_box_pack_start(GTK_BOX(vbox), button, TRUE, TRUE, 2);
gtk_widget_show(button);
/* 将窗口最后显示，以防止屏幕闪烁 */
gtk_widget_show(window);
gtk_main ();
return 0;
}

/*示例结束 */

```

将上面的代码保存为 statusbar.c，然后编写一个如下所示的 Makefile 文件。

```

CC = gcc
statusbar: statusbar.c
    $(CC) `gtk-config --cflags` statusbar.c \
        -o statusbar `gtk-config --libs`
clean:
    rm -f *.o statusbar

```

编译后，运行结果如图 14-1 所示。按下 push item 按钮，向状态条栈中压入一条消息，按 pop last item 按钮，弹出最后一条消息，在状态条上显示下一条消息。

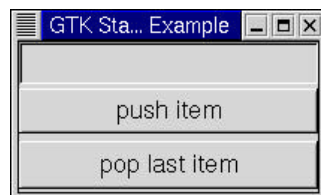


图14-1 状态条示例

第15章 对话框

在Gtk+中，要使用对话框是很麻烦的。每当你要通知用户一些事情，都得创建一个窗口、几个按钮、一个标签，并将按钮、标签等组装到窗口上，然后设置回调函数。同时，还要捕获delete_event事件，等等。Gnome提供了一个容易使用的、通用的对话框构件和几个子构件，用它们可以创建通用对话框。Gnome还有几个使用模态对话框的函数。

15.1 GnomeDialog构件

因为存在于普通Gtk+中的对话框是一种权宜之计，有多少个程序员，就有多少种创建对话框的方法。程序员必须决定对话框放在屏幕的什么地方，构件之间的填充空白大小是多少，是否在按钮之间放置分隔线，按钮放在什么容器中，以及应该设置什么快捷键，等等。GnomeDialog构件的前提就是程序员不应该关心这些事，用户可以用它们想要的方法设置它们。以程序员的观点来说，对话框“能使用就可以了”。

15.1.1 创建对话框

创建GnomeDialog很简单。下面是基本步骤摘要，后面有详细介绍：

- 如果有适合需要的对话框子类，则使用相应的子类，并且可以跳过下面几步。
- 用gnome_dialog_new()函数创建对话框构件。将对话框的标题、每个按钮的名字作为参数传递到函数中去。
- 用需要的内容组装GNOME_DIALOG(dialog)->vbox。
- 规划一下对话框要做什么用。可以为close或clicked信号连接适当的回调函数。关闭对话框时，可以隐藏或销毁对话框，也可以在点击对话框时自动关闭它，或由你自己处理。有多种交互用户与对话框的方法，所以很重要的一点是要确信所选择的设置组合，在不论用户做什么时都能起作用。

用gnome_dialog_new()函数创建新对话框。第一个参数是对话框的标题，后面的参数表是一个以NULL结束的列表，表示要插入到对话框中的按钮。例如，可以像下面这样设置代码：

```
GtkWidget* dialog;  
dialog = gnome_dialog_new(_("My Dialog Title"),  
                           _("OK"),  
                           _("Cancel"),  
                           NULL);
```

上面的代码创建了一个标题为“ My Dialog Title ”，并且带有一个OK按钮和一个Cancel按钮。字符串用_()宏标志它可以被翻译。OK按钮会放在对话框的最左边。

函数列表：创建GnomeDialog构件

```
#include <libgnomeui/gnome-dialog.h>  
GtkWidget* gnome_dialog_new(const gchar* title,  
                             ...)
```

GnomeDialog API将添加的按钮从0开始编号。因为并没有自动创建一个指向按钮的指针，所以后面可以用这些编号引用按钮。在上面的情况中，OK按钮是0号，Cancel按钮是1号。

在上面的例子中，把按钮命名为OK和Cancel按钮。Gnome为常用的按钮提供了一套“内置按钮”。这些按钮保证每个人都使用OK按钮而不是Ok或OK!按钮。它们还让翻译器只翻译常用字符串一次，并且，它们一般在按钮上插入图标，使按钮更富吸引力，也更容易识别。如果可能，应尽量使用内置按钮。

在gnome_dialog_new()函数中使用内置按钮，用内置按钮宏替代按钮名：

```
dialog = gnome_dialog_new(_("My Dialog Title"),
                           GNOME_STOCK_BUTTON_OK,
                           GNOME_STOCK_BUTTON_CANCEL,
                           NULL);
```

Gnome包括许多内置按钮、内置菜单项和内置的像素映射图片。在 libgnomeui/gnome-stock.h.中有这些东西的详细列表。

15.1.2 填充对话框

创建对话框后，就可以在其中放置一些东西。如果只想放一个标签，也许应该使用GnomeMessageBox构件或一些其他函数（例如gnome_ok_dialog()函数），而不是手动创建对话框。填充对话框很简单：

```
GtkWidget* button;
/* ... 假定前面已经创建了对话框dialog... */
button = gtk_button_new_with_label(_("Push Me"));
gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),
                   button,
                   TRUE,
                   TRUE,
                   0);
```

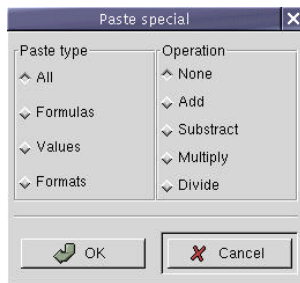


图15-1显示了Gnumeric 电子表格软件中的对话框，它的各部分都加上了标签。

图15-1 Gnumeric 电子表格中的一个GnomeDialog对话框

15.1.3 处理GnomeDialog的信号

创建对话框后，应该能够对用户的动作作出响应。下面是一个可能的动作的列表：

- 按Esc键关闭对话框。
- 点击窗口管理器的关闭按钮关闭对话框。
- 点击其中的某个按钮。
- 与对话框的内容进行交互。
- 如果对话框不是模态的，与应用程序的其他部分进行交互。

除了从它的父类继承的信号外，GnomeDialog还引发另外两个信号。如果用户点击其中的某个按钮，会引发一个 clicked 点击信号（这个信号不是 GtkButton 的 clicked 信号，它是一个不同的信号，由 GnomeDialog 引发）。GnomeDialog 的 clicked 信号处理函数应该有三个参数：引发信号的对话框、被点击的按钮的编号，以及回调数据。

GnomeDialog 还有一个 close 信号。当调用 gnome_dialog_close() 函数时，引发这个信号。

所有的内建处理程序(例如, Esc快捷键)都调用这个函数来关闭对话框。GnomeDialog对close信号的缺省处理程序有两种可能的响应行为:对对话框调用 `gtk_widget_hide()` 或者 `gtk_widget_destroy()` 函数。响应行为可以用 `gnome_dialog_close_hides()` 函数进行配置。

函数列表: 关闭GnomeDialog对话框

```
#include <libgnomeui/gnome-dialog.h>
void gnome_dialog_close_hides(GnomeDialog* dialog,
                              gboolean setting)
```

```
void
gnome_dialog_set_close(GnomeDialog* dialog,
                       gboolean setting)
```

`gnome_dialog_close_hides` 函数设置对话框接收到 close 信号时的行为。如果第二个参数 `setting` 设为 TRUE, 则接收到 close 信号后, 对话框不会关闭, 而是隐藏起来。若设置为 FALSE, 对话框将立即关闭。

`gnome_dialog_set_close` 函数的第二个参数 `setting` 若设置为 FALSE, 对话框将不能关闭, 设置为 TRUE, 则对话框可以关闭。

缺省情况下, close 信号会销毁对话框。通常这也是我们所想要的, 不过, 如果创建对话框很耗时间, 那么可以将它隐藏起来, 使它需要时重新显示, 而不是每次都销毁, 需要时再创建。或许可以将对话框先隐藏起来, 提取出每个构件的状态, 然后用 `gtk_widget_destroy()` 函数销毁它, 具体做法依赖于代码的结构。然而, 通常在点击对话框时销毁它很简单, 不容易出错。要想知道构件在对话框中的状态, 可以连接到 `clicked` 信号。

如果给 close 信号连接一个处理函数, 函数应该返回一个布尔值。如果返回 TRUE, 将不会发生“隐藏”或“销毁”事件。你可以用这种方法阻止用户关闭对话框, 例如, 如果用户还没有填写一个表单中的所有域。

close 信号用于收集用户的几种可能的动作, 并连接到一个单一的处理函数上: 当用户按 Esc 键或点击窗口管理器的关闭按钮时引发该信号。当用户点击对话框上的某个按钮时引发这个信号也会很方便。

注意, 当对话框接受一个 `delete_event` 事件时才引发 close 信号, 这意味着只需要为对话框的所有关闭事件写一个处理函数, 不需要对 `delete_event` 事件单独对待。

15.1.4 最后的修饰

好对话框和伟大的对话框的区别在于细节上。GnomeDialog 对话框带有一些特性, 用它们可以很容易地修饰对话框。下面的函数列表集中介绍了这些函数。

函数列表: 修饰GnomeDialog

```
#include <libgnomeui/gnome-dialog.h>
void gnome_dialog_set_parent(GnomeDialog* dialog,
                             GtkWindow* parent)

void gnome_dialog_set_default(GnomeDialog* dialog,
                              gint button)

void gnome_dialog_editable_enters(GnomeDialog* dialog,
```

```
GtkEditable* editable)
```

```
Void gnome_dialog_set_sensitive(GnomeDialog* dialog,  
    gint button,  
    gboolean setting)
```

对话框有一个逻辑上的双亲，通常是应用程序的主窗口。用 `gnome_dialog_set_parent` 函数设置父子关系，这样 Gnome 会尊重用户的设置，并向窗口管理器指明这种父子关系。大多数窗口管理器会在父窗口最小化时将子窗口也最小化，并让子窗口在父窗口上面。

只能将 `gnome_dialog_set_parent()` 函数和“暂时”对话框一起使用，这一点很重要。“暂时”对话框是指显示和消失相对速度较快的对话框。GnomeDialog 就是一个“暂时”对话框。一些对话框只是一个小窗口，比如说 Gimp 上的工具选项板。这些稳固的（“浮动”）对话框应该可以最小化而无需父窗口最小化，并且它们不应该强制停留在父窗口上。

对话框应该有一个敏感的缺省按钮——当用户按回车键时激活该按钮。用 `gnome_dialog_set_default()` 函数可以指定缺省按钮。应该弄清楚将哪个按钮设置为缺省按钮。通常，最好的选择是最没有破坏性的动作（例如，Cancel 而不是 OK 按钮），但是，如果几个按钮都没有什么破坏性，用户的习惯就是最好的选择。

典型情况下，退出应用程序或删除数据等操作用 Cancel 或 No 按钮作为缺省按钮，要求用户输入文本或其他信息的对话框用 OK 作为缺省按钮。在许多窗口管理器中，当窗口弹出时，该窗口会获得焦点，所以用户想对当前应用程序击键，可实际上却是对对话框的击键。如果对话框将“删除所有文件”选项作为缺省按钮，会有很多人咒骂你。

当在 GtkEditable 构件——编辑构件（及其子构件）上按下回车键时，会引发一个 `activate` 信号。典型情况下，用户期望按回车键激活对话框的缺省按钮，但是如果对话框上有一个编辑构件，例如 `GtkEntry`，它会捕获回车键，对话框的缺省按钮不会对回车响应。当 `GtkEditable` 是激活的时，`gnome_dialog_editable_enters()` 函数激活对话框的缺省按钮，解决了上面的问题。

`gnome_dialog_set_sensitive()` 函数实际上是对其中的按钮调用 `gtk_widget_set_sensitive()` 函数。如果点击某个按钮时什么反应也没有，这个按钮肯定是不敏感的。

最后，应该保证没有创建一个对话框的多个实例。许多应用程序允许弹出多个 Preferences 或 About 对话框。用户并不经常引发这种错误，但是，最好能避免这样的问题。下面的代码用一种简单的方法处理这种问题（注意，创建和显示对话框的代码省略掉了）。

```
void  
do_dialog()  
{  
    static GtkWidget* dialog = NULL;  
  
    if (dialog != NULL)  
    {  
        gdk_window_show(dialog->window);  
        gdk_window_raise(dialog->window);  
    }  
    else  
    {
```

```
dialog = gnome_dialog_new();

gtk_signal_connect(GTK_OBJECT(dialog),
                  "destroy",
                  GTK_SIGNAL_FUNC(gtk_widget_destroyed),
                  &dialog);

/* 此处写显示对话框，连接回调函数等代码 */
}

}
```

`gtk_widget_destroyed()`是在`gtk/gtkwidget.h`中定义的，把它的第二个参数设置为 `NULL` 就可以了。每次用户关闭对话框时，代码会重置对话框变量。如果用户在另一个对话框处活动状态时试图使用它，会将对话框提升为当前窗口，或将对话框从图标中恢复。注意，窗口管理器在提升或者恢复方面有一定的发言权，所以不能保证上面的情况一定会发生。

15.2 模态对话框

有时，当用户与对话框打交道时，要阻止用户与应用程序的其他部分交互。“冻结”了应用程序的其他部分的对话框称为“模态”对话框。

什么时候使用模态对话框还有许多争论。一些用户非常讨厌它，但有许多时候它还是很有必要的。为模态对话框写代码很容易，因为你能够在函数的中间停下来，等着用户做出响应，然后继续。对非模态对话框，必须将控制转交给主应用程序，并安排回调函数，以便用户最后处理对话框时能够回到对话框交出控制权的地方。在一个复杂的对话框中，这样做的结果就是丑陋的面条式的代码。这诱使许多程序员不管什么时候都使用模态对话框，或者至少是经常使用。

如果当用户使用对话框时还想回过头在应用程序主窗口中查阅一些信息，或者用户想在主窗口和对话框之间剪切/粘贴，要避免使用模态对话框。“属性”对话框通常是非模态的，因为用户也许想试验一下他们所做的改变的效果，而不想关闭对话框。没有理由将无足轻重的信息对话框设为模态的，因为在它们上面点击对应用程序的其余部分没有影响。

然而，如果有必要，也不要害怕使用模态对话框。Gnome的文件管理器的“文件属性”对话框是模态的。如果不是这样，用户可能在正在编辑某个文件的属性时将文件删除。这样用户会很困惑。没有什么法则来决定应该使用何种对话框，这完全靠程序员的感觉。

总而言之，创建模态对话框非常容易。在 `Gtk+` 中，任何窗口都可以用 `gtk_window_set_modal()` 设置为模态的。

函数列表：模态窗口

```
#include <gtk/gtkwindow.h>
gtk_window_set_modal(GtkWindow* window,
                    gboolean modality)
```

因为 `GnomeDialog` 是一个 `GtkWindow` 子类，所以你可以用这个函数设置模态窗口。它阻止与模态对话框以外的窗口进行交互。

典型情况下，你可能想更进一步，如等待用户点击对话框的某个按钮，而不用设置多个回调函数。在 `Gtk+` 中这是通过执行 `gtk_main()` 的另一个实例，它是通过进入另一个嵌套的事件循环来实现的。第二个循环退出时，控制权返回到刚才调用的 `gtk_main()` 后面。然而，由于有

大量的关闭对话框的方法，也就有很多的复杂因素和先决条件，最后的代码可能把程序员自己弄糊涂，并且也容易发生错误。下面的两个函数能解决这个问题。

函数列表：“运行”一个对话框

```
#include <libgnomeui/gnome-dialog.h>
gint gnome_dialog_run(GnomeDialog* dialog)
gint gnome_dialog_run and close(GnomeDialog* dialog)
```

执行这两个函数时，程序将停止运行，直到用户点击了一个对话框按钮、点击了窗口管理器的关闭按钮，或者使用快捷键做了同样的事为止。如果点击了某个按钮，它们返回按钮的编号。请注意，GnomeDialog的按钮是从左到右由0开始编号的。如果没有点击按钮(对话框是由窗口管理器关闭的)，它们将返回-1。

在上面的函数调用过程中，对话框自动设置为模态的，否则会引起混乱。例如，从主应用程序代码中调用 `gtk_main_quit()` 函数会退出嵌套的 `gtk_main()`，而不是应用程序的 `gtk_main()`。然而，如果计划在 `gdialog_run()` 返回之后让对话框依然呈打开状态，并且想让它为模态的，应该手动将它设置为模态，`gdialog_run()` 只是临时改变对话框的模态属性。

作为程序员，有责任在调用 `gnome_dialog_run()` 之前解决如何关闭或者销毁对话框问题。你可以设置使用用户的任何动作都不能销毁它，然后在 `gnome_dialog_run()` 返回之后再销毁它。或者，设置对话框使所有的用户对对话框都可以销毁它，然后在 `gnome_dialog_run()` 返回之后将它完全忘记。还可以写一个循环，重复调用 `gnome_dialog_run()` 函数直到用户输入了合法值，并且只有在循环结束之后才能关闭对话框。如果写一个循环，要注意手工将对话框设置为模态，否则它可能会在某个一个较短间隔内是非模态的，这样会引起混乱。

`gnome_dialog_run_and_close()` 监视对话框的 `close` 和 `destroy` 信号，当且仅当对话框没有响应用户点击或击键关闭它时才关闭对话框。使用这个函数保证了 `gnome_dialog_close()` 函数在返回之前只会调用一次。实际上，`gnome_dialog_run_and_close()` 没什么大用处，它不过是一种避免考虑如何关闭对话框的方法而已。

15.3 一个对话框示例

下面是从一个Gnome绘图和制表组件Guppi中摘取的一段代码。这是一个用于“打开”文件的对话框。今后的Gnome版本会有一个GnomeFileSelection构件，对这样的任务它比自己定制一个对话框更合适，但是这里的例子还是有一定的指导作用的。

[illegible]


```

gnome_dialog_editable_enters(GNOME_DIALOG(dialog),
GTK_EDITABLE(gnome_file_entry_gtk_entry(GNOME_FILE_ENTRY(fileentry))));
gnome_dialog_set_default(GNOME_DIALOG(dialog), GNOME_OK);

gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),
                    fileentry,
                    TRUE, TRUE, GNOME_PAD);

gtk_widget_show_all(dialog);

int reply = gnome_dialog_run(GNOME_DIALOG(dialog));

if (reply == GNOME_OK)
{
    gchar* s =
        gnome_file_entry_get_full_path(GNOME_FILE_ENTRY(fileentry),
                                        TRUE);
    /* 此处是应用程序中用于加载文件的详细代码
     * 因为与本节无关，此处略去
     */
}

gtk_widget_destroy(dialog);

```

此示例中调用了 `gnome_dialog_set_close()`，所以如果点击对话框的任何按钮，对话框都会关闭。然而，这里关闭对话框只是调用了 `gtk_widget_hide()` 函数，而不是销毁它。这种行为是由 `gnome_dialog_close_hides()` 函数配置的。`guppi_setup_dialog()` 是 `gnome_dialog_set_parent()` 函数的一个封装，它将应用程序的主窗口设置为对话框的父窗口。

因为对话框的作用是取得一个文件名，按回车键时按下 OK 按钮是很方便的，因而，OK 按钮应该是缺省按钮。但是，通常文本输入框会截取回车键，而 `gnome_dialog_editable_enters()` 正好可以用来修正这个问题。`gnome_dialog_run()` 等待用户采取行动，如果点击“OK”按钮，我们提取文本输入框中的内容，并加载相应的文件。注意，因为调用了 `gnome_dialog_close_hides()` 函数，所以 `gnome_dialog_run()` 返回之后，对话框并不会销毁。然而，`gnome_dialog_run()` 返回之后，对话框会关闭，因为代码保证了所有的用户行为都会关闭它(使用 `gnome_dialog_set_close()` 函数，依靠窗口管理器的关闭按钮的缺省行为)。

最后，`gtk_widget_destroy()` 函数是必要的，因为当关闭对话框时，并没有销毁它。

15.4 特殊对话框

本节介绍一些特殊类型的对话框，它们都是 `GnomeDialog` 的子类，使用这些对话框很方便，并且能够保持用户界面的一致性。当然，前面介绍的关于 `GnomeDialog` 的一些特性也适用于它的子类。

15.4.1 GnomeAbout

Gnome 应用程序应该有一个“关于...”菜单项，用来显示一个关于对话框。Gnome 提供的


```

        NULL);
    gtk_window_set_modal (GTK_WINDOW (about), TRUE);

    gtk_widget_show (about);
}

```

上面的VERSION宏来自于config.h头文件，它是由configure脚本定义的。Gnome Calendar的作者选择将对话框设置为模态来防止多个对话框的实例运行——对话框打开时用户不能再选择菜单项。

15.4.2 GnomePropertyBox——属性框

GnomePropertyBox用于应用程序的参数设置，或者用于编辑用户可见对象的属性。它是一个对话框，内部有一个GtkNotebook构件和四个按钮：OK、Apply、Close、Help。OK按钮等价于点击Apply然后点击Close。放在GnomePropertyBox中的构件用于设置属性或参数值，点击Apply按钮可以让用户请求的改变立即生效。Help按钮用来显示帮助。OK和Close是由属性框自动处理的，所有可以忽略它们。

不需要直接处理属性框的按钮，因为GnomePropertyBox会引发“apply”和“help”信号。这两个信号的处理函数应该是这个样子：

```
void handler(GtkWidget* propertybox, gint page_num, gpointer data);
```

其中page_num是对话框中的GtkNotebook构件中的当前活动页（GtkNotebook是从前往后，从0开始编号的。前页指的是加到GtkNotebook中的第一页）。点击“帮助”按钮时，可以根据页号提供一个上下文敏感的帮助。当用户点击Apply或OK按钮时，对每一页引发一次apply信号，然后最后引发一个信号，并且以-1为page_num值。

要创建一个属性框，首先创建一个对话框，然后创建每一页，添加到对话框中。用gnome_property_box_new()创建GnomePropertyBox，这个函数不带参数。

函数列表：创建GnomePropertyBox

```

#include <libgnomeui/gnome-propertybox.h>
GtkWidget* gnome_property_box_new()
Gint gnome_property_box_append_page(GnomePropertyBox* pb,
                                     GtkWidget* page,
                                     GtkWidget* tab)

```

然后为每一页创建一个构件（可能是一个里面包含许多构件的容器），然后将它用gnome_property_box_append_page()函数添加到属性框中。它的page参数就是放在新笔记本页中的构件，tab参数是用在笔记本标签页上的构件。函数返回新增页的页号，所以不用自己计数。

作为程序员，有责任跟踪用户与每一页内容的任何交互动作。当用户改变了一个设置时，必须通知属性框。当且仅当有变化没有生效时，属性框才会用这些信息设置Apply和OK按钮的敏感性。

属性框状态相关的函数：

```

#include <libgnomeui/gnome-propertybox.h>
void gnome_property_box_changed(GnomePropertyBox* pb)
void gnome_property_box_set_state(GnomePropertyBox* pb,
                                   gboolean setting)

```

gnome_property_box_changed()函数告诉属性框发生的变化。当 apply信号引发时,属性框会自动取消内部的“变化未决”标志。如果需要改变这个内部标志(不太可能会),可以使用 gnome_property_box_set_state()函数。

15.4.3 GnomeMessageBox——消息框

GnomeMessageBox是GnomeDialog的一个子类,它传递一个短消息,或者向用户询问一个简单的问题。Gnome提供了几种类型的消息框,它们的图标和相应的标题在显示的文本前都有所不同。图标可以使用户快速地将显示的消息分类。

GnomeMessageBox的API非常简单,它除了构造函数以外,没有针对 GnomeMessageBox的其他函数。第一个参数是要显示的消息;第二个参数是指定消息框类型的字符串。然后,列出任何按钮,就像 gnome_dialog_new()中的一样。不像未加修饰的 GnomeDialog, 缺省情况下,在 GnomeMessageBox上点击任何按钮都会关闭消息框。当然,可以用 gnome_dialog_set_close()函数改变这种响应方式。

函数列表: 消息框

```
#include <libgnomeui/gnome-messagebox.h>
GtkWidget* gnome_message_box_new(const gchar* message,
                                const gchar* messagebox_type,
                                ...)
```

用下面的宏指定消息框的类型:

- GNOME_MESSAGE_BOX_INFO: 应该用来显示“供参考”的消息。
- GNOME_MESSAGE_BOX_WARNING: 应该用来显示非致命错误。
- GNOME_MESSAGE_BOX_ERROR: 如果操作完全失败,使用这种类型的消息框。
- GNOME_MESSAGE_BOX_QUESTION: 如果要问一个问题,使用这种消息框。
- GNOME_MESSAGE_BOX_GENERIC: 如果上面的都不适用,使用这种消息框。

下面是GnomeMessageBox的用法:

```
GtkWidget * mbox;
mbox = gnome_message_box_new (message,
                              GNOME_MESSAGE_BOX_INFO,
                              GNOME_STOCK_BUTTON_OK,
                              NULL);

gtk_widget_show (mbox);
```

注意 GnomeMessageBox, 像GnomeDialog的大多数子类一样,当点击时它会自动关闭。所以不用手工销毁它。

实用对话框

消息框一般用于向用户显示某些信息,比如提示、警告和错误等。因为消息框差不多总是有同样的按钮(单个OK),Gnome提供了几个实用函数处理这种情况。每一个函数都有一个 _parented()后缀的变体,它调用 gnome_dialog_set_parent()函数设置父窗口。下面的函数列表中的三个函数对分别显示一个信息框、警告框和一个错误框。它们创建并显示对话框,所以,如果愿意可以忽略返回值。

这些函数的唯一目的就是节省打字输入的时间。

第16章 GDK 基础

16.1 GDK和Xlib

GTK是用于实现图形用户接口的函数库。在 Linux平台上，GUI（图形用户接口）使用的是称为X 窗口（X Window）的系统。X窗口系统是1984年由美国麻省理工学院（MIT）开发的。在Linux上使用的X窗口系统是一种称为XFree86的X版本。X窗口系统与Microsoft Windows的图形用户接口有所不同，它是基于客户/服务器的。X服务器在计算机上运行，控制监视器、鼠标和键盘。X客户通过网络与服务器通讯。X服务器为X客户提供图形显示服务。也就是说，X客户和X服务器可能在同一台计算机上运行，也可能在不同的计算机上运行。

X窗口系统带有一套低级的库函数，称为Xlib。Xlib提供了许多对X窗口的屏幕进行操作的函数。当然，使用Xlib函数在屏幕上创建构件是很复杂的。GTK要在屏幕上绘制各种构件，就需要与X服务器打交道。但是GTK提供的构件库并未直接使用Xlib，而是使用了一个称为GDK的库。GDK的意思是GIMP Drawing Toolkit，亦即GIMP绘图工具包。差不多每个Gdk函数都是一个相应Xlib函数的封装。但是Xlib的某些复杂性(和功能)被隐藏起来了。这样是为了简化编程，使Gdk更容易移植到其他窗口系统(有一个在Windows平台上的Gdk版本)。被隐藏的Xlib功能一般是程序员极少用到的，例如，Xlib的许多特性只有窗口管理器才会用到，所以没有封装到Gdk当中。如果需要，可以在应用程序中直接调用Xlib函数，只要在文件头部包含gdk/gdkx.h头文件就可以了。

一般情况下，如果要创建普通的图形接口应用程序，使用GTK就可以了。Gtk+和Gnome构件库提供了极为丰富的构件，足以构造非常复杂的用户界面。但是，如果需要开发新构件，或者要创建绘图程序，仅使用GTK就不够了。这时可以采用Xlib，更好的方法是使用GDK库，它可以应付绝大多数的编程需要。

本章介绍了关于GDK的一些基本知识，这些也是创建构件和绘图的基础。更多的GDK细节内容，请参考gdk.h头文件。

如果了解Gdk函数的实现细节（比如它对应于Xlib的哪一个函数），可以看一下Gdk的源代码以确定它所封装的Xlib函数，然后用man指令参看该函数的手册页。例如，下面是gdk_draw_point()函数的实现代码：

```
void
gdk_draw_point (GdkDrawable *drawable,
                GdkGC      *gc,
                gint        x,
                gint        y)
{
    GdkWindowPrivate *drawable_private;
    GdkGCPrivate *gc_private;

    g_return_if_fail (drawable != NULL);
    g_return_if_fail (gc != NULL);
```



```
drawable_private = (GdkWindowPrivate*) drawable;
if (drawable_private->destroyed)
    return;
gc_private = (GdkGCPrivate*) gc;

XDrawPoint (drawable_private->xdisplay, drawable_private->xwindow,
            gc_private->xgc, x, y);
}
```

每一个数据结构都被转换给它的一个“私有”版本，该“私有”版本包含了与 GDK正在使用的特定窗口系统的相关信息，这样可以将与特定窗口系统相关的函数声明排除在 gdk/gdk.h 头文件外。每个数据结构的“私有”版本都包含一个封装的 Xlib数据结构，且这个数据结构被传递到 XDrawPoint()函数中，所以 XDrawPoint()函数的文档也适用于 gdk_draw_point()函数。

16.2 GdkWindow

GdkWindow是Xlib窗口对象的封装。一个GdkWindow 代表屏幕上的一个区域，可以显示或隐藏起来(在Xlib里面称为映射或反映射窗口)，也可以捕获GdkWindow接收到的事件，还可以在里面绘制图像，移动或调整图像的尺寸。GdkWindow 是以树状结构组织的，也就是说，每一个窗口都可以有子窗口。子窗口是相对于父窗口的位置定位的，当父窗口移动时，子窗口也会移动。子窗口不会在父窗口边界外的区域绘出(也就是说，它们会被父窗口剪裁)。

所谓GdkWindow窗口的树状组织并不是针对每个应用程序的，实际上有一个由 X服务器和窗口管理器控制的窗口的全局树。根窗口没有父窗口，所有窗口都是从它派生而来的。作为桌面背景，根窗口的全部或部分总是可见的。每个窗口都可以为不同的 Linux进程所拥有，一些窗口是由窗口管理器所创建的，还有一些来自于用户的应用程序。

GdkWindow和GtkWindow是完全不同的东西。GtkWindow是一个Gtk+构件，用于表示顶级(toplevel)窗口(顶级窗口是在窗口层次中由应用程序控制的最高级别的窗口)。典型情况下，窗口管理器为顶级窗口创建各种装饰，包括标题条、关闭按钮以及窗口外观等。

要理解X窗口首先要知道它是X服务器上的一个对象，这一点很重要。X客户对每一个窗口获得一个独一无二的整数ID号，并用ID号码引用该窗口。这样，所有的窗口操作都发生在服务器上，并且所有与X窗口打交道的函数都要通过网络传输。

GdkWindow是由X返回的整数ID号的一个封装。它确实保存一些信息的本地拷贝(比如说窗口的尺寸)，所以一些Gdk操作比相应的Xlib操作效率更高。还有，GdkWindow本质上是服务器端对象的一个句柄。许多Gdk对象都是相似的，字体、像素映射图片、鼠标光标等等也是服务器端对象的句柄。

16.2.1 GdkWindow和GtkWidget

许多GtkWidget子类都有一个相关联的GdkWindow窗口。理论上，Gtk+应用程序可以只创建一个顶级窗口，并将所有构件画在里面。然而，这么做并没有什么实际意义，因为GdkWindow允许X窗口系统自动处理许多细节。例如，Gdk所接收到的事件都使用它们所发生的窗口标志，这使得Gtk+能很快确定每个事件是哪个构件发生的。

有一些构件是没有与之相关联的GdkWindow窗口的，它们称为“无窗口”的构件，用一

个GTK_NO_WINDOW标记来标志它们(可以用GTK_WIDGET_NO_WINDOW()宏来测试它们)。没有窗口的构件会将自身绘制在其父构件容器的 GdkWindow窗口上。无窗口构件相对较小, 占用资源较少, 一般将它们称为“轻量级”的, GtkLabel构件就是一个最常见的例子。因为事件总是由GdkWindow窗口接收的, 无窗口构件不能接收事件。如果想让无窗口构件捕获事件, 可以使用GtkEventBox容器构件。

16.2.2 GdkWindow属性

创建GdkWindow时, gdk_window_new() 函数允许指定窗口的所有属性, 这些属性以后也可以再改变。要指定多个属性, 可以向函数传递一个 GdkWindowAttr对象。GdkWindowAttr对象的内容就是GdkWindow窗口的属性。下面是GdkWindowAttr结构的定义:

```
typedef struct _GdkWindowAttr GdkWindowAttr;
struct _GdkWindowAttr
{
    gchar *title;
    gint event_mask;
    gint16 x, y;
    gint16 width;
    gint16 height;
    GdkWindowClass wclass;
    GdkVisual *visual;
    GdkColormap *colormap;
    GdkWindowType window_type;
    GdkCursor *cursor;
    gchar *wmclass_name;
    gchar *wmclass_class;
    gboolean override_redirect;
};
```

因为GdkWindowAttr结构的一些成员是可选的, 所以 gdk_window_new()函数用一个 attributes_mask参数指定哪一个可选成员里包含有效的数据。Gdk 只检查在屏蔽值里面指定的成员, 这样可以让不感兴趣的成员保留缺省值。下面的函数列表简要概括了这些值。没有 attributes_mask标志的成员必须指定, 因为它们没有缺省值。

gdk_window_new()函数最典型的是用在构件的实现中, 用来创建构件的 GdkWindow。在其他场合下极少用到它。gdk_window_destroy()函数销毁GdkWindow窗口。

函数列表: GdkWindow

```
#include <gdk/gdk.h>

GdkWindow* gdk_window_new(GdkWindow* parent,
                          GdkWindowAttr* attributes,
                          gint attributes_mask)

void gdk_window_destroy(GdkWindow* window)
```

下面简要介绍GdkWindowAttr 结构中各个成员的意义。

第一个成员title是GdkWindow的标题, 它只对顶级窗口才有实际意义, 大多数窗口管理器把它放在标题条上。通常, 不要在创建 GdkWindow窗口时指定 title值, 应该让用户调用 gdk_window_set_title()函数来指定它。

第二个成员 `event_mask` 是窗口的事件屏蔽，它决定这个窗口接收什么事件。后面会有详细的介绍。

第三个和第四个成员 `x`、`y` 是窗口的 X、Y 坐标，它们是以像素度量的；这两个坐标是相对于父窗口原点的坐标。每个窗口的原点都是它的左上角（西北角）。注意，它们是 16 位的有符号整数。X 窗口最大可以是 32 768 像素，它可以是负数值，不过会被它的父窗口剪裁掉（只有在父窗口内部的区域才是可见的）。第五个和第六个成员 `width`、`height` 是窗口的宽度和高度，它们是以像素度量的，也是 16 位的有符号整数。

第七个成员 `GdkWindowClass` 可取以下两种值：

- `GDK_INPUT_OUTPUT`：GdkWindow 是一个普通窗口
- `GDK_INPUT_ONLY`：GdkWindow 是一个窗口，它有一个位置，能接收事件，但没有视觉上的表示，也就是说它是不可见的。它的子窗口也必须是这种类型的。你可以为这种窗口设置光标和其他的属性，但是没有办法把窗口画出来（因为它是不可见的）。这种窗口有时候用于捕获事件或改变两个普通窗口重叠区域的鼠标光标。

第八个成员 `visual`（视件）描述了一个显示器的颜色处理特征；第九个成员 `colormap`（颜色表）包含了用于绘画的颜色。

第十个成员 `window_type` 指定 GdkWindow 的类型。窗口可以是下面几种不同类型之一，由 `GdkWindowType` 枚举类型指定：

- `GDK_WINDOW_ROOT`：是根窗口的 Gdk 封装类型，在初始化时创建。
- `GDK_WINDOW_TOPLEVEL`：是一个顶级窗口。在这种情况下，`gdk_window_new()` 函数的 `parent` 参数应该设为 `NULL`。Gdk 自动使用根窗口作为它的父窗口。
- `GDK_WINDOW_CHILD`：是一个在顶级窗口中的下级窗口。
- `GDK_WINDOW_DIALOG`：基本上与顶级窗口是一样的。它的父窗口应该是 `NULL`，并且 Gdk 会替换根窗口。应该设置一个窗口类提示（`wmclass_name`）告诉窗口管理器这个窗口是一个对话框，一些窗口管理器会考虑这些情况并作适当的处理。
- `GDK_WINDOW_TEMP`：用于弹出菜单或其他类似的东西。它是一个只需暂时存在的窗口。它是一个顶级窗口，所以它的父窗口应该是 `NULL`。这种窗口的鼠标光标总是与它们父窗口的一样。所以它们会忽略属性结构中的相关成员值。
- `GDK_WINDOW_PIXMAP`：根本就不是窗口。在 Gdk 中 `GdkPixmap` 和 `GdkWindow` 差不多是同样处理的，所以 Gdk 用同样的结构表示它们；它们可以视为 `GdkDrawable` 类型。
- `GDK_WINDOW_FOREIGN`：标识一个不是由 Gdk 创建的窗口的封装。

对 `gdk_window_new()` 来说，只有 `GDK_WINDOW_TOPLEVEL`、`GDK_WINDOW_CHILD`、`GDK_WINDOW_TEMP` 和 `GDK_WINDOW_DIALOG` 是有效的。库用户不会创建一个 `GDK_WINDOW_ROOT`。`pixmap(GDK_WINDOW_PIXMAP)` 是用 `gdk_pixmap_new()` 创建的。外来窗口 (`GDK_WINDOW_FOREIGN`) 是在 Gdk 外部创建并用 `gdk_window_foreign_new()` 封装的 X 窗口。

第十一个成员 `cursor` 指定在这个窗口中的鼠标的指针（光标）形状。

第十二个成员 `wmclass_name` 前面已经介绍过了。写构件时，通常不设置类提示，它只与顶级窗口有关。Gtk+ 提供了 `gtk_window_set_wmclass()` 函数，程序员可以将它设置为具有确切含义的值。

GdkWindowAttr的最后一个成员 `override_redirect` 决定窗口是否“替换重定向”的。通常，窗口管理器截获所有对顶级窗口的显示、隐藏、移动或尺寸调整请求。你可以重定向或取消这些请求，让顶级窗口按窗口管理器的布局策略行事。将 `override_redirect` 设置为 `TRUE`，禁止窗口管理器对窗口的管理。因为窗口管理器不能移动设置了这个标志的窗口，通常也不会为这样的窗口设置标题条或其他装饰。注意，所有的 `GDK_WINDOW_TEMP` 窗口都将这个成员设为 `TRUE`；`GDK_WINDOW_TEMP` 通常用作弹出菜单，窗口管理器不能控制它。

一般不应该改变 `override_redirect` 成员。如果指定了正确的 `GdkWindowType`，缺省值差不多总是正确的。不过还是有一些例外，例如 Gnome 面板应用程序设置了这个成员。

16.3 视件和颜色表

硬件之间总存在差别。最原始的 X 服务器只支持两种颜色，每一个像素只能是 on 或 off (开或关)。这就是“每像素一位”(bpp)显示模式。每像素一位的显示模式称为深度为 1。多数高级的 X 服务器支持每像素 24 或 32 位，还允许以窗口为基础指定不同的深度。每像素 24 位允许 2^{24} (16 777 216) 种像素，包含了比人眼能分辨的还要多的颜色。

从概念上说，位图显示由一个矩形的像素网格组成。每个像素由一些固定的位数组成；像素以一种硬件相关的方法映射为可视的颜色。考虑这种概念的一种方法就是想象一个二维的整数数组，整数的大小等于要求的位数。换一种说法，可以想象一种显示就像一个“位平面”栈，或“位”的二维数组。如果每个平面都平行于其他平面，那么一个像素就是一根在相同坐标处穿过每个平面的垂线，并且从每一个平面处获得一位。这就是术语“深度”的起源，因为每个像素的位数等于位平面栈的深度。

在 X 窗口系统中，像素代表在一个颜色查找表中的入口。一种颜色就是一组红、绿、蓝 (RGB) 值——监视器以一定比率混合红绿蓝光以显示每个像素。例如，考虑一种八位的显示模式：八位不足以为现实中的颜色编码，只可能为很少部分的 RGB 值编码。作为替代，数据位被解释为整数，用于为 RGB 颜色值做索引。这个颜色表称为 `colormap` (颜色表)，有时，你也可以修改它以包含要使用的颜色，虽然这样做是硬件相关的 (一些 `colormap` 是只读的)。

视件 (visual) 用来决定像素的位模式如何转换为一个可见的颜色。因而，视件还定义了颜色表如何工作。在八位显示中，X 服务器也许将每个像素解释为一个包含 256 种可能颜色值的颜色表的索引。典型情况下，24 位视件有三个颜色表：一个是红色的浓淡值，一个是绿色的浓淡值，一个是蓝色的浓淡值。每个颜色表用一个八位的值索引；三个八位值组成 24 位的像素。视件定义了像素内容的含义，还定义了颜色表是只读的还是可修改的。

简而言之，视件就是特定 X 服务器的颜色容量的描述。在 Xlib 中，你得围绕视件做很多罗嗦的事，但是 Gdk 和 Gtk+ 能极大地简化了这些繁琐工作。

16.3.1 GdkVisual

Xlib 能报告一个所有可用的视件以及相关信息的列表；Gdk 在一个称为 `GdkVisual` 的结构中保持一个这些信息的客户端拷贝。Gdk 能报告可用的视件，并将它们以不同的方式分级。在多数时候，只需用 `gdk_visual_get_system()` 函数就可以了，它返回一个指向缺省视件的指针。如果正在写一个 `GtkWidget` 构件，`gtk_widget_get_visual()` 函数将返回应该使用的视件。返回的

视件不是一个拷贝，所以不需要释放它，Gdk将永久保存视件。

获取缺省视件

```
#include <gdk/gdk.h>
GdkVisual* gdk_visual_get_system()
```

下面是GdkVisual结构的定义，大多数成员都用于由颜色计算像素值。

```
typedef struct _GdkVisual GdkVisual;
struct _GdkVisual
{
    GdkVisualType type;
    gint depth;
    GdkByteOrder byte_order;
    gint colormap_size;
    gint bits_per_rgb;

    guint32 red_mask;
    gint red_shift;
    gint red_prec;

    guint32 green_mask;
    gint green_shift;
    gint green_prec;

    guint32 blue_mask;
    gint blue_shift;
    gint blue_prec;
};
```

16.3.2 视件的类型

视件用不同的度量方法加以区分。它们可以是灰度级或者 RGB值，颜色表可以是可修改的或者固定的，像素值可以是单个的颜色表或者包含了压缩的红、绿、蓝的索引值。下面是GdkVisualType的可能取值：

- GDK_VISUAL_STATIC_GRAY：意味着显示器是单色的或者灰度的，颜色表不能修改。一个像素值只是一个灰度级别，每个像素都是“硬编码”，分别代表一个确定的屏幕上的颜色。
- GDK_VISUAL_GRAYSCALE：意味着显示器是可修改的，但是只有灰度的级别才可能修改。像素代表在颜色表中的一个入口，所以给定的像素在不同的时候能够代表不同的灰度级别。
- GDK_VISUAL_STATIC_COLOR：代表一种颜色显示，它使用单个只读的颜色表而不是红、绿、蓝每种颜色各一个单独的颜色表。这种显示差不多就是 12位或更少(使用单个颜色表的24位显示器需要一个带 2^{24} 个入口的颜色表，差不多有 500MB)。这是一种恼人的视件，因为它的可用颜色太少，而且不能改变成它们的实际颜色。
- GDK_VISUAL_PSEUDO_COLOR：从很多年前开始，就一直是低端PC硬件的常用视件。如果有一个1MB显存、256色的显示卡，这极有可能就是你的 X服务器的视件。它代表一种具有读/写颜色表的颜色显示。像素只对单个颜色表索引。

- **GDK_VISUAL_TRUE_COLOR**：是带三个只读颜色表的颜色显示，红、绿、蓝每种颜色有一个颜色表。一个像素包含三个索引，每个颜色表一个。在像素值和 RGB三元组之间有固定的数学关系，可以用下面的公式从 [0, 255]间的红、绿、蓝值获取像素值：
$$\text{gulong pixel} = (\text{gulong})(\text{red} * 65536 + \text{green} * 256 + \text{blue}).$$
- **GDK_VISUAL_DIRECT_COLOR**：是一种有三个读写颜色表的颜色显示。如果使用 Gdk颜色处理例程，它们只是简单地填充所有三个颜色表以模拟真彩显示。

16.3.3 颜色和GdkColormap

GDK使用GdkColor存储RGB值和像素值。红、绿、蓝值是以 16位无符号整数给出的，取值范围为0到65535。像素的内容依赖于视件。下面是 GdkColor结构定义：

```
typedef struct _GdkColor GdkColor;
struct _GdkColor
{
    gulong pixel;
    gushort red;
    gushort green;
    gushort blue;
};
```

在用一种颜色绘画时，必须：

- 保证像素值包含合适的值。
- 保证颜色值在要使用的可绘区的颜色表中存在（可绘区是一个可以在上面绘画的窗口或 pixmap）。

在Xlib中，这是一个非常复杂的过程，因为需要对每种不同的视件做不同的工作。Gdk对这个过程作了大量的简化。你只需调用 `gdk_colormap_alloc_color()`函数来填充像素值并将颜色值添加到颜色表中即可。下面是一个例子，它使用一个现有的颜色表，这个颜色表应该是要绘画的可绘区的颜色表：

```
GdkColor color;
/* 纯红色 */
color.red = 65535;
color.green = 0;
color.blue = 0;

if (gdk_colormap_alloc_color(colormap, &color, FALSE, TRUE))
{
    /* 成功！ */
}
```

如果`gdk_colormap_alloc_color()`函数返回TRUE，然后分配了一个颜色，`color.pixel`中包含了一个有效的值，这个颜色就可以用于绘画了。`gdk_colormap_alloc_color()`函数中的两个布尔型参数指定了这个颜色是否可写，以及当颜色不能分配时是否尽量找到一个“最匹配”的颜色。如果使用了最匹配的颜色而不是分配一个新颜色，颜色的 RGB值会变成最匹配的值。如果为一个不可写的颜色表请求一个最匹配值，颜色分配不应该会失败。因为即使是在黑白显示器中，黑或者白也会是最好的匹配，只有空的颜色表会导致失败。获得空颜色表的唯一方法就是自己创建一个定制的颜色表。如果不要求最好匹配，在有限颜色数的显示器中极有

可能会失败。在可写的颜色表中很容易会发生失败（此时“最匹配”颜色并没有意义，因为该颜色表能被修改）。

“可写”的颜色表是可以在任何时间改变的颜色表，一些视件支持它，还有一些视件不支持。可写颜色表的作用是改变屏幕的颜色而不用重绘图形。一些硬件在它的颜色查找表中将像素作为索引存储，所以改变查找表就改变了像素的显示。可写颜色表的缺点非常多。最特别的几点：不是所有的视件支持它们，并且可写的颜色表不能由其他应用程序所用，而只读的颜色表可以共享，因为其他的应用程序知道颜色会保持不变。因而，应该尽量避免分配可写的颜色值。在较新的硬件中，使用“可写”颜色表的坏处比获得的好处更多；与直接重绘相比，它并不能显著提高绘图的速度。

使用完一种颜色后，应该用 `gdk_colormap_free_colors()` 函数将它从颜色表中删除。这只对伪彩色和灰度级的视件很重要，在这两种情况下，颜色不够使用，颜色表能被客户修改。Gdk 会自动对每一种视件类型做适当的事情，所以调用这个函数就可以了。

获得RGB值最方便的方法是 `gdk_color_parse()` 函数。这个函数采用 X 颜色规范，填充 `GdkColor` 的红、绿、蓝值。X 颜色规范可以有多种形式，一种可能形式是 RGB 字符串：

```
RGB:FF/FF/FF
```

这指定了一个白色（红绿蓝全部是全亮度）。“RGB:”指定一个“颜色空间”，并决定后面数字的意义。X 系统还能理解其他更晦涩的“颜色空间”。如果颜色规范字符串不是以一个可识别的“颜色空间”开头，X 系统假定它是一个颜色名，并在一个颜色名称数据库中查找。所以可以写下面这样的代码：

```
GdkColor color;
if (gdk_color_parse("orange", &color))
{
    if (gdk_colormap_alloc_color(colormap, &color, FALSE, TRUE))
    {
        /* 得到橙色！*/
    }
}
```

可以看到，如果 `gdk_color_parse()` 函数认出了传给它的字符串，它会返回 `TRUE`。只能使用很少的颜色名，比如“orange”，但是绝大多数颜色都没有相对应的颜色名，也没有办法知道什么名字在数据库里面能找到，所以应该检查函数的返回值。

函数列表：颜色分配

```
#include <gdk/gdk.h>

gboolean gdk_colormap_alloc_color(GdkColormap* colormap,
                                   GdkColor* color,
                                   gboolean writeable,
                                   gboolean best_match)

void gdk_colormap_free_colors(GdkColormap* colormap,
                              GdkColor* colors,
                              gint ncolors)

gint gdk_color_parse(gchar* spec,
                    GdkColor* color)
```

16.3.4 获得颜色表

如果正在写一个 `GtkWidget` 子类，获得颜色表的正确方法就是使用 `gtk_widget_get_colormap()` 函数。另外，系统（缺省）的颜色表通常就是所想要的，调用 `gdk_colormap_get_system()` 函数时不需要参数，它返回缺省的颜色表。

`GdkRGB` 模块是另一种处理颜色的方法，在它的其他功能中，它能用 RGB 值设置图形上下文的背景色和前景色。相关的函数是 `gdk_rgb_gc_set_foreground()` 和 `gdk_rgb_gc_set_background()`。`GdkRGB` 有一个预先分配的颜色表，用来拾取一个最匹配的颜色，使用它意味着应用程序能够与其他使用 `GdkRGB` 的应用程序（比如 GIMP）共享有限的颜色表资源。你还能获得 `GdkRGB` 的颜色表，可以直接使用它。

16.4 可绘区和 pixmap

一个 `pixmap`（像素映射图片）是一个屏幕外的缓冲，可以在里面绘图。当图片存在 `pixmap` 中之后，可以将它复制到一个窗口，当窗口可见时，让它显示在屏幕上。当然，还可以在窗口中绘制。使用 `pixmap` 作为缓冲可以快速更新屏幕而不需要重复一系列原始的绘图操作。`pixmap` 用来存储从磁盘加载的图像数据，比如图标和徽标。然后将图像复制到窗口中。在 `Gdk` 中，`pixmap` 类型称为 `GdkPixmap`。每个像素只有一位 `pixmap` 称为位图，在 `Gdk` 中用 `GdkBitmap` 代表位图。位图（`Bitmap`）并不真是一种单独的类型，从 X 系统的观点来说，它只是一个深度为 1 的 `pixmap`。像窗口一样，`Bitmap` 是服务器端资源。

在 X 的术语学中，一个可绘区就是所有可以在上面绘图的东西。在 `Gdk` 中有一个相应的类型，称为 `GdkDrawable`。可绘区包括窗口、`pixmap` 以及位图。下面是在 `Gdk` 里面的类型定义：

```
typedef struct _GdkWindow GdkWindow;  
typedef struct _GdkWindow GdkPixmap;  
typedef struct _GdkWindow GdkBitmap;  
typedef struct _GdkWindow GdkDrawable;
```

在客户端，`pixmap` 和位图只是类型为 `GDK_WINDOW_PIXMAP` 的 `GdkWindow` 窗口。`GdkDrawable` 用在可接受窗口或者 `pixmap` 为参数的函数声明中。绘制图形的函数使用这两种类型作为参数，移动或者设置“窗口管理器提示”的函数只接受窗口为参数。只有窗口能够接收事件。`GDK_INPUT_ONLY` 窗口是一种特殊情况，它们不是可绘区，不能在上面绘画。

实际上，上面四种可绘区有三种逻辑组合：

	可绘制的	不可绘制的
具有窗口特性	普通窗口	只能输入的窗口
无窗口特性	<code>Pixmap/Bitmap</code>	---

不幸的是，所有这三种逻辑上截然不同的情形从类型检查的观点来看都是一样的。所以要小心使用，不要弄错了。还有，要记住普通窗口在实际显示在屏幕上之前并不是可绘区，应该等到接收到一个“`expose`”事件后才能开始绘画。

像 `GdkWindow` 一样，一个 `GdkPixmap` 仅仅是位于 X 服务器上的对象的客户端句柄。因为这一点，某些操作从性能的观点来看是完全不可行的，例如，如果正在做任何要求对单个像素进行大量操作的事情，可绘区反应会非常慢。另一方面，复制一个 `pixmap` 到窗口中并没有想象的慢，因为两个对象都是在同一台机器上的。

创建一个 pixmap 比创建一个窗口要简单得多，因为大多数窗口属性与 pixmap 是不相关的。用 `gdk_pixmap_new()` 函数创建 pixmap。它以初始尺寸和位深度为参数。如果深度值是 -1，从它的 `GdkWindow` 参数中复制深度。不能为深度随意指定一个值——服务器不支持所有的深度，并且 pixmap 的深度必须与要复制到的窗口的深度一致。调用 `gdk_pixmap_unref()` 函数可以销毁一个 pixmap。

传递到 `gdk_pixmap_new()` 函数中的 `GdkWindow` 参数不是严格需要的。不过，该函数中封装了一个 `XCreatePixmap()` 函数，它以一个 X 窗口作为参数。它用这个参数决定在哪个屏幕上创建窗口，一些 X 服务器有多个显示器。屏幕是一个完全由 Gdk 隐藏起来的 Xlib 概念，Gdk 一次只支持一个屏幕。这样，从 Gdk 的观点来看，`gdk_pixmap_new()` 函数中的 `window` 参数很神秘。

函数列表：GdkPixmap 构造函数

```
#include <gdk/gdk.h>
GdkPixmap*
gdk_pixmap_new(GdkWindow* window,
               gint width,
               gint height,
               gint depth)
void gdk_pixmap_unref(GdkPixmap* pixmap)
```

16.5 事件

事件传送到应用程序中，以指明在一个 `GdkWindow` 中的变化或者有意义的用户动作。所有的事件都与一个 `GdkWindow` 相关联。它们也与一个 `GtkWidget` 相关联，Gtk+ 主循环将事件从 Gdk 传递给 Gtk+ 构件树。

16.5.1 事件类型

事件 GDK 中有多种类型，可以使用 `GdkEvent` 联合体代表任何类型。有一个特殊的事件类型 `GdkEventAny`，它包含了所有事件都有的三个成员，任何事件都可以转换为 `GdkEventAny`。`GdkEventAny` 中的第一个成员是一个类型标志，`GdkEventType`。`GdkEvent` 联合体中也包含 `GdkEventType`。下面是 `GdkEventAny` 结构定义：

```
struct _GdkEventAny
{
    GdkEventType type;
    GdkWindow* window;
    gint8 send_event;
};
```

下面是 `GdkEvent` 联合体的定义：

```
union _GdkEvent
{
    GdkEventType type;
    GdkEventAny any;
    GdkEventAny any;
    GdkEventExpose expose;
    GdkEventNoExpose no_expose;
```

```

GdkEventVisibility      visibility;
GdkEventMotion          motion;
GdkEventButton          button;
GdkEventKey             key;
GdkEventCrossing        crossing;
GdkEventFocus           focus_change;
GdkEventConfigure       configure;
GdkEventProperty        property;
GdkEventSelection        selection;
GdkEventProximity       proximity;
GdkEventClient          client;
GdkEventDND             dnd;
};

```

每个事件类型都以 GdkEventAny 中的三个成员作为它的头三个成员。这样，有多种方式引用一个事件的类型（假设 GdkEvent* 称为一个事件）：

```

event->type
event->any.type
event->button.type
((GdkEventAny*)event)->type
((GdkEventButton*)event)->type

```

在 Gtk+ 的源代码中有可能看到上面的各种方法。当然，每个事件子类型都有它自己唯一的成员，type 成员指出什么子类型是有效的。

GdkEventAny 的 window 成员是事件要传送到的 GdkWindow 窗口。如果 send_event 标志是 TRUE，事件就会由另一个应用程序（或本应用程序）合成；如果是 FALSE，事件由 X 服务器引发。Gdk 并不传送事件的 X 接口（XSendEvent()）。不过，Gtk+ 经常通过声明一个静态的 event 结构来“虚构”一个事件，在结构中填充相应的值，然后引发与事件相对应的构件信号。这些合成的事件中 send_event 设置为 TRUE。

对 GdkEventType，有比 GdkEvent 联合体中更多的可能值。许多事件类型共用同样的数据。例如，因为当鼠标按键按下和弹起时，会传递同样的信息，GDK_BUTTON_PRESS 和 GDK_BUTTON_RELEASE 都使用 GdkEvent 的 button 成员。表 16-1 显示了 GdkEventType 枚举类型和相应的 GdkEvent 成员中所有的可能取值。每个事件的类型在本节稍后介绍。

表16-1 GdkEventType值

值	GdkEvent成员
GDK_NOTHING	None
GDK_DELETE	GdkEventAny
GDK_DESTROY	GdkEventAny
GDK_EXPOSE	GdkEventExpose
GDK_MOTION_NOTIFY	GdkEventMotion
GDK_BUTTON_PRESS	GdkEventButton
GDK_2BUTTON_PRESS	GdkEventButton
GDK_3BUTTON_PRESS	GdkEventButton
GDK_BUTTON_RELEASE	GdkEventButton
GDK_KEY_PRESS	GdkEventKey
GDK_KEY_RELEASE	GdkEventKey
GDK_ENTER_NOTIFY	GdkEventCrossing
GDK_LEAVE_NOTIFY	GdkEventCrossing

(续)

值	GdkEvent成员
GDK_FOCUS_CHANGE	GdkEventFocus
GDK_CONFIGURE	GdkEventConfigure
GDK_MAP	GdkEventAny
GDK_UNMAP	GdkEventAny
GDK_PROPERTY_NOTIFY	GdkEventProperty
GDK_SELECTION_CLEAR	GdkEventSelection
GDK_SELECTION_REQUEST	GdkEventSelection
GDK_SELECTION_NOTIFY	GdkEventSelection
GDK_PROXIMITY_IN	GdkEventProximity
GDK_PROXIMITY_OUT	GdkEventProximity
GDK_DRAG_ENTER	GdkEventDND
GDK_DRAG_LEAVE	GdkEventDND
GDK_DRAG_MOTION	GdkEventDND
GDK_DRAG_STATUS	GdkEventDND
GDK_DROP_START	GdkEventDND
GDK_DROP_FINISHED	GdkEventDND
GDK_CLIENT_EVENT	GdkEventClient
GDK_VISIBILITY_NOTIFY	GdkEventVisibility
GDK_NO_EXPOSE	GdkEventNoExpose

16.5.2 事件屏蔽

每个GdkWindow窗口都有一个与之相关联的事件屏蔽，它决定 X服务器将哪一个事件传递到应用程序。你可以在创建 GdkWindow窗口时将它作为 GdkWindowAttr结构的一部分指定事件屏蔽。以后还可以用 gdk_window_set_events()和gdk_window_get_events()函数存取或改变事件屏蔽。如果要设置的 GdkWindow属于一个构件，则不应该直接改变事件屏蔽，应该使用gtk_widget_set_events()或gtk_widget_add_events()函数。gtk_widget_set_events() 函数在构件已经实现后才能使用，它可以在任何时候用 gtk_widget_add_events()函数向一个已存在的事件屏蔽添加事件。

函数列表：GdkWindow事件屏蔽

```
#include <gdk/gdk.h>
GdkEventMask gdk_window_get_events(GdkWindow* window)
void gdk_window_set_events(GdkWindow* window,
                           GdkEventMask event_mask)
```

函数列表：构件的事件屏蔽

```
#include <gtk/gtkwidget.h>
gint gdk_widget_get_events(GtkWidget* widget)
void gtk_widget_add_events(GtkWidget* widget,
                           gint event_mask)
void gtk_widget_set_events(GtkWidget* widget,
                           gint event_mask)
```

表16-2显示了什么事件要求什么样的事件屏蔽值。有些事件不用指定就会接收到，特别是：

- Map, unmap, destroy和configure事件是用GDK_STRUCTURE_MASK指定的，但是对

任何新窗口，Gdk都会自动选中它们。不过，Xlib并不这样做。

- 选择、拖放以及删除事件没有事件屏蔽，因为它们会被自动选中（对所有的窗口，Xlib都选中它们）。

表16-2 事件与屏蔽类型

事件屏蔽	屏蔽事件类型
GDK_EXPOSURE_MASK	GDK_EXPOSE
GDK_POINTER_MOTION_MASK	GDK_MOTION_NOTIFY
GDK_POINTER_MOTION_HINT_MASK	N/A
GDK_BUTTON_MOTION_MASK	GDK_MOTION_NOTIFY(鼠标键按下时)
GDK_BUTTON1_MOTION_MASK	GDK_MOTION_NOTIFY(鼠标左键按下时)
GDK_BUTTON2_MOTION_MASK	GDK_MOTION_NOTIFY(鼠标右键按下时)
GDK_BUTTON3_MOTION_MASK	GDK_MOTION_NOTIFY(鼠标中间键按下时)
GDK_BUTTON_PRESS_MASK	GDK_BUTTON_PRESS
GDK_BUTTON_RELEASE_MASK	GDK_BUTTON_RELEASE
GDK_KEY_PRESS_MASK	GDK_KEY_PRESS
GDK_KEY_RELEASE_MASK	GDK_KEY_RELEASE
GDK_ENTER_NOTIFY_MASK	GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY_MASK	GDK_LEAVE_NOTIFY
GDK_FOCUS_CHANGE_MASK	GDK_FOCUS_IN, GDK_FOCUS_OUT
GDK_STRUCTURE_MASK	GDK_CONFIGURE
GDK_PROPERTY_CHANGE_MASK	GDK_PROPERTY_NOTIFY
GDK_VISIBILITY_NOTIFY_MASK	GDK_VISIBILITY_NOTIFY
GDK_PROXIMITY_IN_MASK	GDK_PROXIMITY_IN
GDK_PROXIMITY_OUT_MASK	GDK_PROXIMITY_OUT
GDK_SUBSTRUCTURE_MASK	对子窗口接收到GDK_STRUCTURE_MASK
GDK_ALL_EVENTS_MASK	所有事件

16.5.3 在Gtk+中接收Gdk事件

在一个Gtk+程序里面不会直接接收Gdk事件。相反，所有事件都是传递到一个GtkWidget构件，由它引发一个相应的信号。通过连接一个信号处理函数到信号上来处理事件。

X服务器给每个X客户发送一个事件流。事件按它们发生的次序被发送和接收。Gdk将它接收到的每一个XEvent转换为一个GdkEvent事件，然后将事件放在一个队列里面。对每个被接收的事件来说，它决定哪一个构件（如果有的话）接收事件。GtkWidget基类对大多数事件类型（比如说button_press_event）都定义了信号；它也定义了一个普通的“事件”信号。Gtk+主循环调用gtk_widget_event()函数将事件传递给构件，这个函数首先引发一个“事件”信号，然后只对特定事件类型（如果合适）引发一个信号。一些事件是用特殊的方法处理的；例如，drag-and-drop(拖放)事件并不是直接对应着drag-and-drop信号。

通常，构件的事件属于事件所发生的GdkWindow。不过，还有下面几种特殊情况：

- 如果构件具有独占性（也就是说，如果调用了gtk_grab_add()函数），某些事件只会转发给具有独占性的构件，或者这个构件的子构件。发生在其他构件上的事件会被忽略。只有由用户引发的事件，比如鼠标按键事件和键盘事件才会受独占影响。
- 构件的敏感性也会影响事件传递的位置。用户交互事件不会传递到不敏感的构件上。

可以预见，没有相关联的GdkWindow窗口的构件（比如GtkLabel）不会产生事件；X只

给窗口传递事件。有一个例外：容器为它们的无窗口子构件合成 expose 事件。

Gtk+主循环将事件从子构件传给它们的父容器。也就是，对每个事件，信号先由它的子构件引发，然后是它紧挨着的父构件，然后是父构件的父构件，等等。依次向上递归。例如，如果点击一个 GtkMenuItem，它会忽略按钮按下事件，让它的上层菜单处理它。有一些事件是不会传播的。

如果一个构件“处理”了事件，事件传播就会结束。这保证了对任何用户动作，只会发生一个用户可见的变化。构件的事件信号处理程序必须返回一个 gint 型整数值。最后一个要运行的信号处理程序决定了信号引发的返回值。所有的事件信号都是 GTK_RUN_LAST 类型的，所以返回值来自于下面几种：

- 如果有的话，用 gtk_signal_connect_after() 连接的最后一个处理程序。
- 否则，构件的缺省信号处理程序，如果有的话。
- 否则，用 gtk_signal_connect() 连接的信号处理程序(如果有)。
- 否则，缺省返回值是 FALSE。

如果一个信号引发返回值是 TRUE，Gtk+主循环会停止当前事件的传播。如果它返回 FALSE，主循环会将事件传播给它的父构件。每个构件都会导致两种信号引发：一种普通的“事件”信号和一种特殊的信号(比如 button_press_event 或 key_press_event)。如果两种引发都返回 TRUE，事件传播将停止。普通“事件”信号的返回值还有例外的效果：如果是 TRUE，第二个信号(特殊信号)不会引发。

表16-3概括了 GtkWidget 信号与事件类型的对应关系。该事件受活动的“鼠标独占”的影响，并沿父构件到子构件的路径传播。所有事件信号处理程序都返回一个 gint 类型的整数值，带有三个参数：引发信号的构件、触发信号的事件、用户数据指针。

表16-3 GtkWidget 事件

事件类型	GtkWidget 信号	是否传播	是否独占
GDK_DELETE	"delete_event"	否	否
GDK_DESTROY	"destroy_event"	否	否
GDK_EXPOSE	"expose_event"	否	否
GDK_MOTION_NOTIFY	"motion_notify_event"	是	是
GDK_BUTTON_PRESS	"button_press_event"	是	是
GDK_2BUTTON_PRESS	"button_press_event"	是	是
GDK_3BUTTON_PRESS	"button_press_event"	是	是
GDK_BUTTON_RELEASE	"button_release_event"	是	是
GDK_KEY_PRESS	"key_press_event"	是	是
GDK_KEY_RELEASE	"key_release_event"	是	是
GDK_ENTER_NOTIFY	"enter_notify_event"	否	是
GDK_LEAVE_NOTIFY	"leave_notify_event"	否	是*
GDK_FOCUS_CHANGE	"focus_in_event", "focus_out_event"	否	否
GDK_CONFIGURE	"configure_event"	否	否
GDK_MAP	"map_event"	否	否
GDK_UNMAP	"unmap_event"	否	否
GDK_PROPERTY_NOTIFY	"property_notify_event"	否	否
GDK_SELECTION_CLEAR	"selection_clear_event"	否	否
GDK_SELECTION_REQUEST	"selection_request_event"	否	否
GDK_SELECTION_NOTIFY	"selection_notify_event"	否	否

(续)

事件类型	GtkWidget信号	是否传播	是否独占
GDK_PROXIMITY_IN	"proximity_in_event"	是	是
GDK_PROXIMITY_OUT	"proximity_out_event"	是	是
GDK_CLIENT_EVENT	"client_event"	否	否
GDK_VISIBILITY_NOTIFY	"visibility_notify_event"	否	否
GDK_NO_EXPOSE	"no_expose_event"	否	否

* 如果正被“独占”的构件接收到了相应的GDK_ENTER_NOTIFY事件，GDK_LEAVE_NOTIFY事件会发生在该“独占”的构件。这保证了“进入”事件（GDK_ENTER_NOTIFY）和“离开”事件（GDK_LEAVE_NOTIFY）是成对发生的。

16.5.4 鼠标按键事件

在GdkEventButton中有四种事件类型：

- GDK_BUTTON_PRESS 表明一个鼠标按键被按下。
- GDK_BUTTON_RELEASE 表明一个鼠标按键按下后按键被释放了。不一定是按下鼠标的地方发生这个事件，如果用户将鼠标移动到一个不同的 GdkWindow窗口，新的窗口会接收这个事件，而不是按下鼠标的窗口。
- GDK_2BUTTON_PRESS 表明鼠标按键在一个较短的时间内连续点击了两次：双击事件。该事件总是在GDK_BUTTON_PRESS和GDK_BUTTON_RELEASE事件的第一次点击之后发生。
- GDK_3BUTTON_PRESS表明鼠标键在很短时间间隔内连续点击了三次：“三击”事件。该事件总是在两个 GDK_BUTTON_PRESS/GDK_BUTTON_RELEASE事件对和GDK_2BUTTON_PRESS之后发生。

如果在同一个GdkWindow窗口上快速点击了三次鼠标，下面的事件会按次序接收到：

- GDK_BUTTON_PRESS
- GDK_BUTTON_RELEASE
- GDK_BUTTON_PRESS
- GDK_2BUTTON_PRESS
- GDK_BUTTON_RELEASE
- GDK_BUTTON_PRESS
- GDK_3BUTTON_PRESS
- GDK_BUTTON_RELEASE

当鼠标按键被按下时，X服务器自动发生指针独占，释放按键时，解除指针独占。这意味着按键释放事件总是发生在接收“按键按下”事件的同一个窗口上。Xlib允许改变这种行为，但是Gdk不允许（在Xlib文档中，这种自动独占称为“被动独占”。这与用gdk_pointer_grab()函数主动独占窗口是截然不同的）。

按键事件是像下面这样定义的：

```
typedef struct _GdkEventButton GdkEventButton;
struct _GdkEventButton
{
```

```

GdkEventType type;
GdkWindow *window;
gint8 send_event;
guint32 time;
gdouble x;
gdouble y;
gdouble pressure;
gdouble xtilt;
gdouble ytilt;
guint state;
guint button;
GdkInputSource source;
guint32 deviceid;
gdouble x_root, y_root;
};

```

按钮事件在X服务器中是用一个时间标记来标志的。时间是按服务器端时间、以毫秒度量的，每隔几周，整数就会溢出，然后时间标记就会从0重新开始。因而，不应该依赖于绝对的时间值；它一般用于判定事件之间的相对时间。

GdkEventButton结构中包含的x、y是鼠标指针相对于事件发生的窗口的x和y坐标。记住，鼠标指针可能会在窗口的外边(如果指针独占有效)。如果鼠标在窗口的外边，坐标值可能是负值或比窗口的尺寸还大。坐标是以双精度值给出的，而不是整数，因为一些输入设备，例如图形输入板有比像素还小的度量单位。对大多数情况，可能要将双精度值转换为整数。一些输入设备还有pressure(压力)、xtilt(x方向斜移)以及ytilt(y方向斜移)等特征，大多数情况下都可以忽略它们。

GdkEventButton的state成员指示在按钮按下之前哪个组合键或鼠标按键是按下的。它是一个位域，设置为下表中的一个或多个标志值。因为组合键值是在按键按下之前读出的，它只会使用在“按键按下”之前的state的值，但是“按键释放”事件不是这样的。

应该细心检查确切的位屏蔽值，而不是检查state域的精确值。也就是说，尽量使用这样的代码：

```
if ( (state & GDK_SHIFT_MASK) == GDK_SHIFT_MASK )
```

避免使用这样的代码：

```
if ( state == GDK_SHIFT_MASK )
```

如果检查state成员的精确值，如果应用程序的Num Locks键或其他较难注意到的按键是打开的，应用程序有可能不会按所设想的方式工作，并且也很难检查出问题所在。

表16-4 鼠标按键和键盘按键的修改键屏蔽值

事 件	修改键含义
GDK_SHIFT_MASK	Shift键
GDK_LOCK_MASK	CapsLock键
GDK_CONTROL_MASK	Ctrl键
GDK_MOD1_MASK	Mod1(通常是Meta或Alt键)
GDK_MOD2_MASK	Mod2键
GDK_MOD3_MASK	Mod3键
GDK_MOD4_MASK	Mod4键

(续)

事 件	修改键含义
GDK_MOD5_MASK	Mod5键
GDK_BUTTON1_MASK	鼠标按键1
GDK_BUTTON2_MASK	鼠标按键2
GDK_BUTTON3_MASK	鼠标按键3
GDK_BUTTON4_MASK	鼠标按键4
GDK_BUTTON5_MASK	鼠标按键5
GDK_RELEASE_MASK	键盘按键释放

GdkEventButton 的button成员指明是哪个按键触发了事件(也就是,哪个按键被按下或释放)。按键是从1到5编号的,大多数场合,按键1是左键,按键2是中间键,按键3是右键。左撇子也许会将按键颠倒过来。按键4和按键5事件是由滚轮鼠标在旋转滚轮时生成的,Gtk+会捕获这些事件并移动旁边的滚动条。有的鼠标没有中间键,也没有滚轮,因而,最好不要写一些依赖于按键2、4或5引发的事件的程序。

三个标准的鼠标键在Gnome中有约定的含义。按键1用于选择、拖放以及操作构件等最常见的任务。按键3用于激活一个弹出菜单。按键2通常用于移动对象,比如面板。有时按键1也用于移动对象,例如桌面图标可以用按键1或按键3移动。应该尽量与其他应用程序保持一致。

GdkEventButton 的source和deviceid成员用于决定是哪一个设备触发了事件,例如,用户也许同时连接了一块图形输入板和一个鼠标。除非要写一个能利用非鼠标输入设备的应用程序,可以忽略这两个成员。

GdkEventButton 的最后两个成员,x_root和y_root,是相对于根窗口的x和y的坐标,而不是相对于接收事件的窗口的坐标。可以用这些“绝对”坐标来比较来自于两个不同窗口的事件。

16.5.5 键盘事件

只有两种类型的键盘事件:GDK_KEY_PRESS和GDK_KEY_RELEASE。一些硬件不会产生“放开按键”事件;不应该写依赖于GDK_KEY_RELEASE事件的代码,虽然如果有构件接收到该事件,代码也会正常反应。

下面是键盘事件的类型定义:

```
typedef struct _GdkEventKey GdkEventKey;
struct _GdkEventKey
{
    GdkEventType type;
    GdkWindow *window;
    gint8 send_event;
    guint32 time;
    guint state;
    guint keyval;
    gint length;
    gchar *string;
};
```

前三个成员是GdkEventAny事件中的标准成员,time和state成员与GdkEventButton中的一样的。

第六个成员 `keyval` 包含一个键符。X 服务器保留了一个全局的翻译表，它将实际按键和控制键的组合转换成键符。例如，键盘上的“ A ”键，产生不带控制键的 `GDK_a` 键符，和按下 `shift` 键的 `GDK_A` 键符。用户可以改变物理按键和键符之间的映射关系，例如，可以重新排列按键，创建一个 Dvorak 键盘(更普通的情况：可以将 `Ctrl` 和 `Caps Lock` 键交换，或将 `Alt` 键用作 `Meta` 键)。键符是在 `gdk/gdkkeysyms.h` 头文件中定义的。如果要使用 `keyval` 成员，应该包含这个文件。键符是用字符串来表示的。例如，`GDK_a` 键符映射为字符串“ a ”。不过，X 服务器允许修改键符到字符串的映射。

`GdkEventKey` 的 `string` 成员包含一个键符的字符串描述，`length` 成员包含字符串的长度。`length` 值可能是 0 (缺省状态，许多非字母的按键并没有字符串描述)。

通常，如果为了将用户打的字显示出来而读取键盘事件，应该使用 `GdkEventKey` 的 `string` 成员。例如，`GtkEntry` 和 `GtkText` 构件都使用了 `string` 成员。字处理程序也应该使用这个成员。如果因为其他原因而读取键盘事件(比如说键盘快捷键)，或者对缺省没有字符串描述的按键(比如说功能键或方向键)感兴趣，需要使用在 `gdk/gdkkeysyms.h` 头文件里面定义的 `keyval` 和 `keysym`。

下面是一个简单的按键事件回调函数示例，它演示了怎样从按键事件中获取信息。对任何构件，将这个回调函数连接到 `key_press_event` 信号都是很合适的：

```
static gint
key_press_cb(GtkWidget* widget, GdkEventKey* event, gpointer data)
{
    if (event->length > 0)
        printf("The key event's string is '%s\n", event->string);
    printf("The name of this keysym is '%s",
           gdk_keyval_name(event->keyval));
    switch (event->keyval)
    {
        case GDK_Home:
            printf("The Home key was pressed.\n");
            break;
        case GDK_Up:
            printf("The Up arrow key was pressed.\n");
            break;
        default:
            break;
    }

    if (gdk_keyval_is_lower(event->keyval))
    {
        printf("A non-uppercase key was pressed.\n");
    }
    else if (gdk_keyval_is_upper(event->keyval))
    {
        printf("An uppercase letter was pressed.\n");
    }
}
```

`gdk_keyval_name()` 函数对调试很有用，它返回不带 `GDK_` 前缀的键符。例如，如果向它

传递的是 `GDK_Home`，则返回 “ `Home` ”。字符串是静态分配内存的。如果键符是大写的，`gdk_keyval_is_lower()`函数将返回 `FALSE`。对小写字母、数字以及所有的非字母的字符，它返回 `TRUE`；它只对大写字母返回 `FALSE`。`gdk_keyval_is_upper()`与`gdk_keyval_is_lower()`刚好相反。

16.5.6 鼠标移动事件

当鼠标在屏幕上移动时，可以使用鼠标移动事件跟踪它的移动。移动事件是当鼠标指针在窗口内移动时发生的，穿越事件是在鼠标指针进入或离开 `GdkWindow`窗口时发生的。移动事件中的典型成员是 `GDK_MOTION_NOTIFY`。有两种类型的穿越事件：`GDK_ENTER_NOTIFY`和`GDK_LEAVE_NOTIFY`。

有两种方法跟踪鼠标移动事件。如果在窗口的事件屏蔽中指定了 `GDK_POINTER_MOTION_MASK`，可以接收到X服务器能产生的尽可能多的事件。如果用户快速移动指针，程序会被移动事件淹没，必须快速处理它们，否则应用程序在处理大量事件时会反应迟钝。如果还指定了 `GDK_POINTER_MOTION_HINT_MASK`，那么每次只发送一个移动事件。也要调用`gdk_window_get_pointer()`函数、鼠标指针离开并重新进入窗口，或者鼠标按键或键盘事件发生时，事件才会发送出去。也就是，每次接收到一个移动事件必须调用`gdk_window_get_pointer()`函数取得鼠标指针的当前位置，并通知X服务器已经可以接收另一个事件了。

选择何种模式依赖于应用程序。如果需要精确跟踪指针的轨迹，就得捕获所有的移动事件。如果仅仅关心最近的鼠标指针位置，为了将网络流量最小化，使应用程序的响应能力最大化，则应在应用程序的事件屏蔽中包含 `GDK_POINTER_MOTION_HINT_MASK`。`gdk_window_get_pointer()`函数要求在服务器和客户之间传输数据以获得鼠标指针的位置，所以它对应用程序的响应能力有很大的限制。如果能尽可能快地处理鼠标运动事件而不被大量的事件淹没，应用程序看起来会比没有设置事件屏蔽 `GDK_POINTER_MOTION_HINT_MASK` 要快些。鼠标移动事件不可能在一秒钟内发生 200次，所以如果能在5毫秒内处理好鼠标移动事件，情况就会不错。

如果想在一个月或多个鼠标按键按下时才接收鼠标事件，可以用 `GDK_BUTTON_MOTION_MASK`代替 `GDK_POINTER_MOTION_MASK`。还可以用 `GDK_POINTER_MOTION_HINT_MASK`和`GDK_BUTTON_MOTION_MASK`一起使用来限制要接收的事件的数量，就像和`GDK_POINTER_MOTION_MASK`一起使用一样。如果仅仅对某个特定的鼠标按键按下时的鼠标移动事件感兴趣，可以使用与特定鼠标键相关的事件屏蔽 `GDK_BUTTON1_MOTION_MASK`、`GDK_BUTTON2_MOTION_MASK`以及 `GDK_BUTTON3_MOTION_MASK`。这三个事件屏蔽的任何组合都是允许的。它们也可以与`GDK_POINTER_MOTION_HINT_MASK`联合使用以限制事件发生的数量。

总而言之，可以用下面五个事件屏蔽值选择在什么按键状态下接收什么鼠标移动事件：

- `GDK_POINTER_MOTION_MASK`：不管按键状态，接收所有事件。
- `GDK_BUTTON_MOTION_MASK`：有按键按下时接收所有事件。
- `GDK_BUTTON1_MOTION_MASK`：按键1按下时接收所有事件。
- `GDK_BUTTON2_MOTION_MASK`：按键2按下时接收所有事件。

• GDK_BUTTON3_MOTION_MASK: 按键3按下时接收所有事件。

缺省状态下，应用程序会被 X 服务器能生成的事件所淹没，最好在事件屏蔽中添加一个 GDK_POINTER_MOTION_HINT_MASK，让事件“一段时间内发生一次”。

移动事件用 GdkEventMotion 结构描述：

```
typedef struct _GdkEventMotion GdkEventMotion;
struct _GdkEventMotion
{
    GdkEventType type;
    GdkWindow *window;
    gint8 send_event;
    guint32 time;
    gdouble x;
    gdouble y;
    gdouble pressure;
    gdouble xtilt;
    gdouble ytilt;
    guint state;
    gint16 is_hint;
    GdkInputSource source;
    guint32 deviceid;
    gdouble x_root, y_root;
};
```

大多数成员与 GdkEventButton 中的成员是类似的。事实上，唯一与 GdkEventMotion 不同的成员是 is_hint 标志。如果它是 TRUE，GDK_POINTER_MOTION_HINT_MASK 标志会选中。如果要写一个供其他人使用的构件，并且想让他们选择怎样接收移动事件，需要设置这个成员的值。在移动事件处理程序中，可以这么做：

```
double x, y;
x = event->motion.x;
y = event->motion.y;
if (event->motion.is_hint)
    gdk_window_get_pointer(event->window, &x, &y, NULL);
```

也就是说，只在必要时调用 gdk_window_get_pointer() 函数。

如果正在使用 GDK_POINTER_MOTION_HINT_MASK，给定事件的坐标应该使用从 gdk_window_get_pointer() 函数取得的值，因为它们是更新过的。如果正在接收每个事件，调用 gdk_window_get_pointer() 就没有什么意义，因为这样做太慢了，并且会引起严重事件积压——这样，最终会得到所有的事件，但是应用程序的性能会很糟糕。

穿越事件是在鼠标指针进入或离开一个窗口时发生的。如果在应用程序的 GdkWindow 之间快速移动，Gdk 会为每一个被穿越的窗口产生穿越事件。不过，Gtk+ 会试图删除在多个“穿越”过程中的事件，只将第一个离开窗口事件和最后一个进入窗口事件传给构件。这种优化能够提高程序的响应速度。如果感觉到应该发生了进入 / 离开事件，可实际上并没有发生，可能是上面的优化造成的。

GdkEventCrossing 结构的定义如下：

```
typedef struct _GdkEventCrossing GdkEventCrossing;
struct _GdkEventCrossing
```

```
{
    GdkEventType type;
    GdkWindow *window;
    gint8 send_event;
    GdkWindow *subwindow;
    guint32 time;
    gdouble x;
    gdouble y;
    gdouble x_root;
    gdouble y_root;
    GdkCrossingMode mode;
    GdkNotifyType detail;
    gboolean focus;
    guint state;
};
```

上面所示结构中的很多成员都应该是很熟悉的，其中 x 和 y 坐标是相对于发生穿越事件的窗口的坐标； x_root 和 y_root 坐标是相对于根窗口的； $time$ 成员指明事件的时间； $state$ 成员指明发生事件时按下的鼠标按键和组合键。结构中的前三个成员是 `GdkEventAny` 中的三个标准成员。不过，这里还有几个新成员。

`GdkEventCrossing`结构中的 $window$ 成员是一个指向鼠标指针要进入或离开的窗口指针， x 和 y 坐标就是相对于这个窗口的。但是，在“离开”事件发生之前，鼠标指针也许已经在接收事件的窗口的子窗口中存在着；当“进入”事件发生时，鼠标指针也许在一个子窗口中消失了。在这些情况下， $subwindow$ 成员应该设置为这个子窗口。否则，可以将 $subwindow$ 设置为`NULL`。注意，如果在子窗口的事件屏蔽值中设置有 `GDK_ENTER_NOTIFY_MASK`或`GDK_LEAVE_NOTIFY_MASK`，子窗口也会接收到它自己的“进入”或者“离开”事件。

`GdkEventCrossing` 中的 $mode$ 成员指出事件是正常引发的还是作为指针独占的一部分。当指针被独占或解除独占时，指针也许会移动。由一个独占引起的鼠标移动事件的 $mode$ 成员值为`GDK_CROSSING_GRAB`，由解除独占引起的移动事件的 $mode$ 成员为`GDK_CROSSING_UNGRAB`，所有其他情况 $mode$ 都为`GDK_CROSSING_NORMAL`。

`GdkEventCrossing` 中的 $detail$ 成员很少用到。它给出了关于要离开和进入的窗口在 X 系统窗口树中的相对位置的一些信息。它有两个很简单、很有用的值：

- `GDK_NOTIFY_INFERIOR` 标志一个当指针移进或移出一个子窗口时，由父窗口接收到的穿越事件。
- `GDK_NOTIFY_ANCESTOR` 标志一个当指针移进或者移出一个父窗口时，由子窗口接收到的穿越事件。

其他几种值也是可能的：`GDK_NOTIFY_VIRTUAL`，`GDK_NOTIFY_INFERIOR`，`GDK_NOTIFY_NONLINEAR`，`GDK_NOTIFY_NONLINEAR_VIRTUAL`以及`GDK_NOTIFY_UNKNOWN`。不过，它们绝不会用到，因为它们太复杂了。

键盘焦点

`GdkEventCrossing`的 $focus$ 成员指出是事件窗口还是它的子窗口得到键盘输入焦点。键盘焦点是一个 X 概念，用于决定哪一个窗口应该接收按键事件。窗口管理器决定哪一个顶级窗口应该获得焦点。通常，获得焦点的窗口是高亮显示的，并在最前面显示。大多数窗口管理器会让你在“跟随鼠标获得焦点”和“点击获得焦点”间作出选择。当应用程序具有焦点时，

可以将焦点在它的子窗口间自由移动（例如，在不同的 GtkText 构件间移动）。不过，Gtk+ 并不对子窗口使用 X 的焦点机制。顶级 GtkWidget 构件是唯一接收 X 焦点的窗口。因而，它们从 X 服务器（通过 Gdk）接收所有的原始按键事件。Gtk+ 实现了它自己的构件焦点概念，它和 X 的窗口焦点概念是类似的，但实际上是截然不同的。当一个顶级窗口接收到按键事件，它会将事件传给具有 Gtk+ 焦点的构件。

简而言之，如果包含事件窗口的顶级 GtkWidget 窗口目前具有 X 焦点，focus 成员的值就是 TRUE。Focus 成员与 Gtk+ 的构件焦点概念没有直接的关系。

16.5.7 焦点变更事件

前面直接解释了 Gtk+ 和 X/Gdk 的键盘焦点概念的区别。只有一种类型的焦点事件，GDK_FOCUS_CHANGE，当一个窗口或获得或失去键盘焦点，就会发生这个事件。从 Gdk 的观点来看，只有顶级 GtkWidget 构件才能获得或失去焦点，这样这个事件好象没什么用。不过，每个 GtkWidget 窗口构件维护一个当前“焦点构件”，并将键盘事件传递给这个构件。当焦点构件改变时，它也合成一个 Gdk 风格的焦点事件。这样，即使没有用到 Gdk 风格的焦点事件，构件以用到这种事件相同的方法接收到焦点。有一点细微的区别：不论构件的 GtkWidget 的事件屏蔽值是否设置包含了 GDK_FOCUS_CHANGE_MASK，它都会接收到焦点变更事件。只有顶级构件需要指定这个屏蔽值。

下面是焦点事件 GdkEventFocus 结构的定义。焦点事件本身很简单。当一个构件获得键盘焦点，它接收到一个焦点事件，同时它的 in 成员设置为 TRUE，当它失去焦点时，它也接收到这个焦点事件，它的 in 成员设置为 FALSE。除此之外，焦点事件只包含 GdkEventAny 中的三个标准成员：

```
typedef struct _GdkEventFocus GdkEventFocus;
struct _GdkEventFocus
{
    GdkEventType type;
    GtkWidget *window;
    gint8 send_event;
    gint16 in;
};
```

16.6 鼠标指针

鼠标在屏幕上用一个称为“光标”的位图显示。普通情况下光标是一个箭头形状，但是它可以以窗口为基础改变。鼠标指针移动时，它产生移动事件，并在屏幕上移动光标，向用户反馈信息。

16.6.1 指针定位

可以用 gdk_window_get_pointer() 函数查询指针的位置。这个函数以指针的 x 和 y 坐标作为参数；坐标是相对与第一个参数“window”的。它还获得当前的活动修改键（包括修改键和按钮；这个成员与几个其他事件，比如按钮事件中 state 成员是一样的）。如果给 x, y 或 state 参数传递 NULL 值，这个参数会被忽略。

函数列表：查询指针位置

```
#include <gdk/gdk.h>
GdkWindow* gdk_window_get_pointer(GdkWindow* window,
                                   gint* x,
                                   gint* y,
                                   GdkModifierMask* state)
```

16.6.2 独占指针

可以独占指针，这意味着在独占期间，所有的鼠标指针事件都会发生在独占的窗口上。通常指针事件都发生在指针所在的窗口上。例如当用户正在用“点击 - 拖动”来选择一个矩形的区域时，应该独占指针。如果点击鼠标后将指针拖到窗口外边，应该继续跟踪鼠标的位置，并据此改变选择的范围。独占也保证了指针事件不会发送到其他的应用程序中。

调用gdk_pointer_grab()函数独占指针。函数的第一个参数是被独占的窗口，在独占期间，这个窗口将接收到事件。下一个参数应该是TRUE或FALSE，它指定事件是否只发生在独占的窗口，或者还包括它的子窗口。confine_to参数指定一个限制指针的窗口。用户不能将指针移出这个窗口。在独占期间，可以指定不同的光标形状。如果不想改变光标，将cursor设置为NULL。因为光标是一种服务器端资源，直到独占结束，X不会释放它，调用完gdk_pointer_grab()后可以立即销毁光标。

最后一个参数time指定独占何时生效，它是指服务器时间。它主要用于解决当两个客户试图同时独占指针时的冲突，time必须在上次独占之后，并且不能是在将来的某个时间。通常，总是将time成员设置为正在处理事件的时间，或者使用GDK_CURRENT_TIME宏。GDK_CURRENT_TIME让X服务器代入当前时间。

如果成功，gdk_pointer_grab()返回TRUE。如果独占窗口或confine_to是隐藏的，另一个客户已经独占，或者任何参数是无效的时，它也有可能失败。遗憾的是很少有应用程序检查这个返回值，这是一个缺陷。不过，因为很难触发这个错误，忽略这个问题一般不会引起严重后果。

调用gdk_pointer_ungrab()函数可以取消指针独占，参数time和gdk_pointer_grab()中的time参数一样。可以用gdk_pointer_is_grabbed()函数来检查指针是否被独占了。当使用完之后，必须将其取消独占，否则在指针被独占时用户不能使用其他的应用程序。

注意，Gdk级的独占指针和Gtk+级的独占指针的概念是不一样的。Gtk+中的独占将某些事件重定向到具有独占构件，生成一个“模态”构件，比如对话框。Gtk+的独占只影响当前的应用程序，只有发生在当前应用程序的某个构件上的事件才会被重定向。Gdk的独占范围更宽，包含了整个X服务器，而不仅仅是一个应用程序。

函数列表：独占指针

```
#include <gdk/gdk.h>
gint gdk_pointer_grab(GdkWindow* window,
                      gint owner_events,
                      GdkWindow* confine_to,
                      GdkCursor* cursor,
                      guint32 time)

void gdk_pointer_ungrab(guint32 time)

gint gdk_pointer_is_grabbed()
```


16.6.3 改变光标

在任何时候都可以改变光标的形状，光标形状是用 `gdk_window_set_cursor()` 函数设置的，它与特定的窗口相关。缺省时，窗口使用它们父窗口的光标。将窗口的光标设置为 `NULL`，就可以恢复它的缺省光标。

有两种方法创建光标。最简单的方法就是从 X 中的光标字体中挑选一个。光标字体包含光标而不是字符，可以用 `xfd -fn` 命令察看这些光标字体。每个光标形状在 `gdk/gdkcursors.h` 中都定义了一个常数。`gdk_cursor_new()` 可以接受这种常数作为它唯一的参数：

```
GdkCursor* cursor;  
cursor = gdk_cursor_new(GDK_CLOCK);  
gdk_window_set_cursor(window, cursor);  
gdk_cursor_destroy(cursor);
```

注意，一旦将光标连接到窗口，就可以销毁光标了。`GdkCursor` 是服务器端资源的客户端句柄，只要它还在使用，X 就会保持服务器端资源。

如果光标字体中没有一种适合需要，可以用位图创建一个定制的光标。实际上需要两个位图：源 pixmap 和屏蔽位图。因为它们是位图，每个像素都是 on 或 off (0 或 1)。如果在屏蔽位图中像素是 1，这个像素点会是透明的。如果像素在 pixmap 中也是 1，它会以传递到 `gdk_cursor_new_from_pixmap()` 函数中的 fg (前景色) 颜色显示。如果像素点在屏蔽位图中是 1，但是在源 pixmap 中是 0，它会以 bg (背景色) 颜色显示。源和屏蔽 pixmap 的大小必须是一样的，它们的深度必须都是 1。

前景色和背景色应该明暗对比强烈，这样光标才会在任何背景上都是可见的。大多数光标都用前景色绘制，用背景色作为轮廓 (要看到这一点，在黑色的背景上移动 X 光标，可以看到在边缘上是白色的轮廓)。要做到这一点，屏蔽位图应该和源 pixmap 形状一样，尺寸稍大一些。

`gdk_cursor_new_from_pixmap()` 函数的最后两个参数是光标热点的坐标。这个点应该是鼠标指针指向点的位置——箭头光标的尖端，或者十字交叉光标的中心。如果光标热点不在位图内部，调用 `gdk_cursor_new_from_pixmap()` 函数创建光标会失败。

函数列表：`GdkCursor`

```
#include <gdk/gdk.h>  
  
GdkCursor* gdk_cursor_new(GdkCursorType cursor_type)  
  
GdkCursor* gdk_cursor_new_from_pixmap(GdkPixmap* source,  
                                       GdkPixmap* mask,  
                                       GdkColor* fg,  
                                       GdkColor* bg,  
                                       gint x,  
                                       gint y)  
  
void gdk_cursor_destroy(GdkCursor* cursor)  
  
void gdk_window_set_cursor(GdkWindow* window,  
                           GdkCursor* cursor)
```

16.7 字体

X 字体是一种服务器端资源。本质上来说，字体是位图的集合，这些位图代表了不同的字

符。一种字体中不同字符的位图具有相似的尺寸和风格。Gdk允许用一种称为GdkFont的客户端句柄来操纵字体。

要获得一种GdkFont, 调用gdk_font_load() 函数(或者使用先前存在的GtkStyle中的字体)。字体是用字体名加载的, 字体名是一个相当棘手的话题。字体名遵从称为“X 逻辑字体描述”或XLFD的约定。要了解XLFD的最好的方法是使用随X发布的xfontsel实用程序。还可以用xlsfonts 程序得到一个X服务器上的字体名列列表。

字体名是一个用连字号分隔开的域组成的字符串。每个域描述了字体的一些方面的信息。例如:

```
-misc-fixed-medium-r-normal--0-0-75-75-c-0-iso8859-1
```

或

```
-adobe-new century schoolbook-bold-i-normal--11-80-100-100-p-66-iso8859-1
```

十四个域分别是:

Foundry: 字体供应商的名字, 例如 Adobe 或 Sony。对随X发布的普通字体, 用 misc。

Family: 字体的字样或风格, 例如 Courier、Times、Helvetica 等等。

Weight: 粗体、半粗体、一般体等。

Slant: 斜体、罗马体、倾斜体(缩写为 I, r, 或 o)。

Set Width: 字体的“比例宽度”, 包括普通、紧缩、半紧缩等几种可能取值。

Add Style: 字体的任何附加信息都可以放在这里, 它用于区别名字相同的两种字体。此处的字符串没有限制。

Pixels: 字体的像素数目。

Point: 字体的点数, 以十分之一为基准。一点是 1/72 英寸, 点和像素的关系是由 X 服务器所识别的显示器的分辨率(每英寸的字点数)决定的。典型情况, 人们并不配置 X 服务器以适应监视器的特性, 所以 X 服务器对当前分辨率的设置也许是不准确的。

Horizontal Resolution: 水平显示分辨率, 以要显示的字体的每英寸字点数计量。

Vertical Resolution: 字体的垂直分辨率。

Spacing: 可为两种取值 - Monospace (缩写为 m) 或 proportional (缩写为 p)。指明所有的字符都是相同的宽度或是不同的字符可以有不同的宽度。

Average Width: 在字体中所有字体的平均宽度, 以 1/10 像素计。

Character Set Registry: 定义字符集的组织或标准。

Character Set Encoding: 指定具体字符集的编码方式。最后两个域结合起来指定了字符集。对欧洲语言来说总是使用 iso8859-1。这是“Latin-1”字符集, 它是一种八位编码, 其中包含 ASCII 作为它的子集。

使用字体时, 不必指定所有 14 个域。可以使用“通配符”: * 代表任意个字符; ? 代表任意一个字符。例如, 160 点粗体 Roman Helvetica 字体可以像下面这样:

```
-*-helvetica-bold-r-*-*-160-*-*-*-*-*
```

当传递字体名到 gdk_font_load() 函数时, 应该只将它作为一种缺省选择。其他国家的用户可能想使用适合于他们的语言的字体, 美国和欧洲用户可能也想自己定义字体。还有, 有些字体在其他服务器也许并不存在。因而, 应该提供一种方法来定制要用到的任何字体。最容易的方法就是使用从构件的 GtkStyle 中取得的字体。

如果没有找到与所给字体名匹配的字体，`gdk_font_load()`函数会返回NULL。当使用过一种字体后，应该调用`gdk_font_unref()`函数释放它。

加载字体时，至少要指定字体名、`weight`、`slant`、`size`等属性——否则，通配符*也许会随机加载一个粗体、斜体的字体，这可能不是实际需要的。Xlib编程手册建议总是要指定字体的点数，这样用户在不同的监视器上都能得到正确的显示效果。不过，X服务器不一定能正确识别出显示分辨率，所以这样做的理论意义多于实际意义。也许更好的方法是指定像素值，因为可以知道所显示的其他元素的像素大小。没有完美的方法，最好使应用程序的字体是可配置的。

函数列表：GdkFont

```
#include <gdk/gdk.h>
GdkFont* gdk_font_load(const gchar* font_name)
void gdk_font_unref(GdkFont* font)
```

字体规格

要使用字体，典型情况下需要知道字体规格的详细信息。字体规格用于字符之间的定位，以及决定用字体绘制字符串时的大小。最基本的规格是字体的上行和下行。文本放在一条基线上，基线就像笔记本的纸上的一把尺子。每个字符的底部都挨着基线。一些字符（比如小写的“p”和“y”）向基线下延伸。字体的下行就是字符在基线下的最大距离。它的上行值就是在基线之上的距离。字体的高度是上行值和下行值的和。绘制多行字体时，至少得在每条基线间留出字体高度这样的距离。

在GdkFont结构定义中有上行(`ascent`)和下行(`descent`)成员：

```
typedef struct _GdkFont GdkFont;
struct _GdkFont
{
    GdkFontType type;
    gint ascent;
    gint descent;
};
```

第一个成员`type`将字体从字体集中区别开来；字体集用于显示非欧洲语言。

在字体中，个别字符有自己的上行值和下行值；字符的上行值和下行值必须小于或等于字体的上行值和下行值。Gdk能计算特定字符串的最大上行值和下行值的和，而不是计算整个字体的。该高度应该小于或等于字体的高度。相关的函数是 `gdk_string_height()`、`gdk_text_height()`和`gdk_char_height()`。`gdk_text_height()`函数与`gdk_string_height()`略有不同，前者接受字符串的长度作为一个参数，而 `gdk_string_height()`直接调用`strlen()`。如果已经知道了字符串的长度，应该使用`gdk_text_height()`函数。

除了垂直规格，在字体中，每个字符还有三个描述它的水平尺寸的规格。字符的 `width`是从一个字符的左边起点到下一个字符开始的距离。注意，`width`不是到字符左边起点到最右边像素点的距离，在一些字体中，特别是斜体中，字符可能会倾斜超过下一个字符的起点。`left-side-bearing`或`lbearing`是从字符左边起点到最左边像素点的距离；`right-side-bearing`或`rbearing`是从字符起点到最右边像素点的距离。因而，`rbearing`可能会比`width`值还大。在斜体字中，字符倾斜越过了下一个字符的起始点。

所有返回字符或字符串宽度的Gdk函数都返回字符的宽度，或者返回在字符串内字符宽度

的总和。如果最右边的字符的rbearing值比它的宽度大，字符串也许需要比gdk_string_width(), gdk_text_width(), 或gdk_char_width()所返回值更多的空间。对测量高度的函数，_string_函数变体计算字符串的长度，而_text_变体函数接受一个预先计算好的长度作为参数。

通常，以_measure结尾的函数才是编程所需要的。对一个有N个字符串的函数，这些函数返回前N - 1个字符的宽度之和，加上最后一个字符的rbearing值。也就是，它们会考虑rbearing值也许会比width值大。如果要决定为绘制一个字符串留多少空间，可能要用gdk_string_measure()、gdk_text_measure()或gdk_char_measure()函数。不过，有时不应该考虑字符的rbearing值，例如，如果要居中对齐字符串，也许使用字符的宽度值更合适（因为如果不使用宽度，一个很小的、超过字符串宽度的斜体装饰并不会“填充”空白区域，字符串看起来会略为中央靠左）。

gdk_text_extents()和gdk_string_extents()返回字符串的所有规格，包括lbearing、rbearing、width、ascent和descent等。函数返回的left-side-bearing是字符串最左边像素点的，right-side-bearing是最右边的像素点的，和gdk_text_measure()返回的一样。返回的width值是字符宽度的总和，和gdk_text_width()返回的值一样。

所有的字体规格都是在客户端计算的，所以与其他的绘图函数相比，这些函数占用资源还不算昂贵。

函数列表：字体规格

```
#include <gdk/gdk.h>
gint gdk_string_width(GdkFont* font,
                     const gchar* string)
gint gdk_text_width(GdkFont* font,
                   const gchar* string,
                   gint string_length)
gint gdk_char_width(GdkFont* font,
                   gchar character)
gint gdk_string_measure(GdkFont* font,
                      const gchar* string)
gint gdk_text_measure(GdkFont* font,
                    const gchar* string,
                    gint string_length)
gint gdk_char_measure(GdkFont* font,
                    gchar character)
gint gdk_string_height(GdkFont* font,
                     const gchar* string)
gint gdk_text_height(GdkFont* font,
                    const gchar* string,
                    gint string_length)
gint gdk_char_height(GdkFont* font,
                    gchar character)
void gdk_string_extents(GdkFont* font,
                      const gchar* string,
                      gint* lbearing,
                      gint* rbearing,
                      gint* width,
                      gint* ascent,
```

```

        gint* descent)
void gdk_text_extents(GdkFont* font,
        const gchar* string,
        gint string_length,
        gint* lbearing,
        gint* rbearing,
        gint* width,
        gint* ascent,
        gint* descent)

```

16.8 图形上下文

一个图形上下文，或者GC (Graphics Context)，是一套在绘图时要用到的参数(比如颜色、剪裁屏蔽值、字体等等)。它是一种服务器端资源，就像 pixmap和窗口一样。GC减少了Gdk绘图函数的参数个数，也减少了每个绘图请求从客户到服务器间传递的参数数目。

与GdkWindowAttr类似，图形上下文可以用GdkGCValues结构创建。结构中包含了图形上下文中所有的特性，还可以传递gdk_gc_new_with_values()标志指出哪一个成员是有效的。其他的成员保留其缺省值。还可以用gdk_gc_new()函数(这种方法通常更容易)创建一个全为缺省值的GC。创建GC之后，还有一些函数用来改变GC的设置，但是要记住，每次改变GC设置值都需要一条消息传递到X服务器。

所有的GC都是不可以互换的，它们都与特定的深度和视件相关联。GC的深度和视件必须与要绘图的可绘区的深度和视件相匹配。GC的深度和视件是从传递到gdk_gc_new()函数的GdkWindow*参数中获得的，所以处理这种问题的最容易的方法就是在要绘图的窗口上创建GC。

下面是GdkGCValues结构的定义：

```

typedef struct _GdkGCValues GdkGCValues;
struct _GdkGCValues
{
    GdkColor          foreground;
    GdkColor          background;
    GdkFont           *font;
    GdkFunction       function;
    GdkFill           fill;
    GdkPixmap         *tile;
    GdkPixmap         *stipple;
    GdkPixmap         *clip_mask;
    GdkSubwindowMode  subwindow_mode;
    gint              ts_x_origin;
    gint              ts_y_origin;
    gint              clip_x_origin;
    gint              clip_y_origin;
    gint              graphics_exposures;
    gint              line_width;
    GdkLineStyle      line_style;
    GdkCapStyle       cap_style;
    GdkJoinStyle      join_style;
};

```

前景色 (foreground成员) 是画线、圆或其他形状时的“画笔颜色”。背景色 (background成员)

的用处依赖于特定的绘画操作。这些颜色必须是用`gdk_color_alloc()`函数在当前颜色表中分配的。

`font`成员没有用到：在 Xlib 中绘制文本时用它指定字体。在 GdK 以前的版本中，它也有同样的作用；但是新的绘制文本的 GdK 程序都要求一个 `GdkFont*` 参数。一个 Xlib 图形上下文只能存储无格式的字体，但是 `GdkFont` 能够代表一个字体集（用以绘制一些外语文字）。

`function` 成员指定要画的像素点与可绘区上已有的像素点如何结合起来。有许多种可能取值，但是只有两种是最常用的：

- `GDK_COPY`：缺省值。它忽略已存在的像素点（只是将新的像素点画在上面）。
- `GDK_XOR`：将旧的和新的像素点一种可反转的方式结合起来。也就是，如果执行两次 `GDK_OR` 操作，头一次绘图就会被第二次操作取消。`GDK_XOR` 通常用于“擦除”，可以恢复可绘区原来内容。

`GdkGCValues` 的 `fill` 成员决定如何使用 `GdkGCValues` 中的 `tile` 和 `stipple` 成员。其中 `tile` 成员是一个与目的可绘区深度相同的 `pixmap` 图片，它被反复复制到目的可绘区，将它们拼贴起来——第一次拼贴的原点是 `(ts_x_origin, ts_y_origin)`。而 `stipple` 成员是一个位图（深度为 1 的 `pixmap`）；它也是从 `(ts_x_origin, ts_y_origin)` 开始拼贴的。下面是 `fill` 的可能取值：

- `GDK_SOLID`：忽略 `tile` 和 `stipple` 成员。绘图形状是用前景色和背景色绘制的。
- `GDK_TILED`：绘图形状用 `tile` 成员指定的 `pixmap` 图片绘制，而不是用前景色和背景色。用 `GDK_TILED` 模式绘画会擦除可绘区上的任何内容，显示由 `tile` 成员指定的图片的拼贴图形。
- `GDK_STIPPLED`：用 `stipple` 中定义的位绘制图形。也就是，在 `stipple` 成员中未设置的位不会绘出。
- `GDK_OPAQUE_STIPPLED`：用前景色绘制在 `stipple` 中设置的位，没有在 `stipple` 中设置的位用背景色绘制。

有些 X 服务器并没有有效实现上面所有的 `fill` 模式值，所以使用时可能会很慢。

`clip_mask` 成员是可选的，它是一个位图，只有在这个位图中设置了的位才会画出。从 `clip_mask` 到可绘区的映射是由 `clip_x_origin` 和 `clip_y_origin` 值决定的，这些定义了与 `clip_mask` 中 (0,0) 对应的可绘区的坐标。也可以设置一个剪裁矩形（最常用的、也是最有用的形式）或一个剪裁区域（区域就是在屏幕上的任意范围，典型情况是一个多边形或矩形列表）。用 `gdk_gc_set_clip_rectangle()` 设置剪裁矩形：

```
GdkRectangle clip_rect;
clip_rect.x = 10;
clip_rect.y = 20;
clip_rect.width = 200;
clip_rect.height = 100;
gdk_gc_set_clip_rectangle(gc, &clip_rect);
```

要关闭“剪裁”，将剪裁的矩形、区域或剪裁屏蔽值设置为 `NULL`。

GC 的 `subwindow_mode` 只与可绘区是否为一个窗口有关。缺省设置是 `GDK_CLIP_BY_CHILDREN`；这意指子窗口不会被在父窗口上的绘图影响。这会造成一个假象：子窗口在父窗口的“上面”，并且子窗口是不透明的。`GDK_INCLUDE_INFERIORS` 在所有在“上面”的子窗口上绘制，改写子窗口上包含的任何图形——通常不使用这种模式。如果确实在使用 `GDK_INCLUDE_INFERIORS` 模式，可能要使用 `GDK_XOR` 作为绘图函数，因为它允许恢复子窗口原先的内容。

graphics_exposures是一个布尔值，缺省是TRUE，它决定gdk_window_copy_area()是否产生expose事件。

GC的最后四个值决定怎样画线。这些值用于画线，包括未填充多边形的边框以及弧线。line_width域决定线的宽度(以像素计)。宽度为0的线称为一条“细线”，细线是一个像素宽的线，绘制得非常快(通常使用硬件加速)，但是画的具体像素依赖于所使用的X服务器。为了一致性，最好使用宽度为1的线。

line_style域可以是下面三种值：

- GDK_LINE_SOLID是缺省值；一条实线。
- GDK_LINE_ON_OFF_DASH用前景色画一条虚线，将虚线的 off(关闭)部分空着。
- GDK_LINE_DOUBLE_DASH用前景色画一条虚线，但是虚线的 off(关闭)部分用背景色绘制。

虚线是gdk_gc_set_dashes()指定的；GdkGCValues中并不包括这个域。gdk_gc_set_dashes()需要三个参数：

- dash_list是一个虚线长度的数组。偶数号的长度是“on”(打开)部分，它们是用前景色绘制的；奇数号的长度是“off”(关闭)部分，它们不画出，或者用背景色绘制，具体绘制方法依赖于line_style。长度值不能是0，所有的值必须是正数。
- dash_offset是在虚线列表中第一个像素的索引号。也就是，如果在dash_list中指定了5个on和5个off，并且dash_offset是3，绘制的线将从第3个on虚线开始。
- N是在dash_list中的元素的个数。

可以设置一个古怪的虚线模式，例如：

```
gchar dash_list[] = { 5, 5, 3, 3, 1, 1, 3, 3 };  
gdk_gc_set_dashes(gc, 0, dash_list, sizeof(dash_list));
```

缺省的dash_list是{4, 4}，偏移量是0。

图16-1显示了一些用GDK_LINE_DOUBLE_DASH画的虚线。图形上下文的前景色是黑色，背景色是亮灰色。头5根线是缺省的{4, 4}虚线模式，偏移量分别是0、1、2、3和4。记住，缺省值是0。图16-2显示了这5根线的放大图。最后的一根线就是上面提到的古怪虚线模式，它的放大图显示在图16-3。



图16-1 5根虚线，线型为GDK_LINE
_DOUBLE_DASH

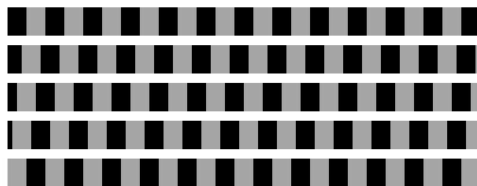


图16-2 缺省的虚线模式，偏移量不同

cap_style决定X怎样画线的端点(或虚线端点)。它有4种可能取值：

- GDK_CAP_BUTT：是缺省值，它意味着线的端点是正方形的。

- **GDK_CAP_NOT_LAST**：对应一个像素宽度的线，最后一个像素忽略不画。其他与 **GDK_CAP_BUTT**一样。
- **GDK_CAP_ROUND**：在线的端点画一个小弧线，由线的端点向两边延伸。弧线的中心是线的端点，半径是线宽的一半。对一个像素宽的线，它没有什么效果（因为没有办法画一个像素宽的弧线）。
- **GDK_CAP_PROJECTING**：将线延伸，越过它的终点半个线宽。它对一个像素的线没有效果。



图16-3 复杂的虚线模式

join_style参数影响画多边形或在一个函数中画多条线时，各线之间如何连接。如果把线想象成一个细长的矩形，就很容易弄清楚线之间并不是平滑连接的。在连接的两个端点之间有一个凹槽。对这个凹槽有三种处理方法，也就是 **join_style**的三种可能取值：

- **GDK_JOIN_MITER**：是缺省值，在线交叉的地方画一个尖角。
- **GDK_JOIN_ROUND**：在交叉的凹槽处画一个弧线，画一个圆形的转角。
- **GDK_JOIN_BEVEL**：用最小的可能的形状填充凹槽，画一个平坦的转角。

函数列表：GdkGC

```
#include <gdk/gdk.h>
GdkGC* gdk_gc_new(GdkWindow* window)
GdkGC* gdk_gc_new_with_values(GdkWindow* window,
                              GdkGCValues* values,
                              GdkGCValuesMask values_mask)
void gdk_gc_set_dashes(GdkGC* gc,
                      gint dash_offset,
                      gchar dash_list,
                      gint n)
void gdk_gc_unref(GdkGC* gc)
```

GC 属性

属性	GdkGCValuesMask	Modifying	函 数	缺省值
GdkColor	foreground	GDK_GC_FOREGROUND	<code>gdk_gc_set_foreground()</code>	黑色
GdkColor	background	GDK_GC_BACKGROUND	<code>gdk_gc_set_background()</code>	白色
GdkFont	*font	GDK_GC_FONT	<code>gdk_gc_set_font</code>	依赖于X服务器
GdkFunction	function	GDK_GC_FUNCTION	<code>gdk_gc_set_function()</code>	GDK_COPY
GdkFill	fill	GDK_GC_FILL	<code>gdk_gc_set_fill()</code>	GDK_SOLID
GdkPixmap	*tile	GDK_GC_TILE	<code>gdk_gc_set_tile()</code>	用前景色填充的 Pixmap
GdkPixmap	*stipple	GDK_GC_STIPPLE	<code>gdk_gc_set_stipple()</code>	所有位都是on(打 开)的位图
GdkPixmap	*clip_mask	GDK_GC_CLIP_MASK	<code>gdk_gc_set_clip_mask()</code>	none
GdkSubwindowMode	Subwindow_mode	GDK_GC_SUBWINDOW	<code>gdk_gc_set_subwindow()</code>	GDK_CLIP_BY _CHILDREN

(续)

属性	GdkGCValuesMask	Modifying	函 数	缺省值
Gint	ts_x_origin	GDK_GC_TS_X_ORIGIN	gdk_gc_set_ts_origin()	0
Gint	ts_y_origin	GDK_GC_TS_Y_ORIGIN	gdk_gc_set_ts_origin()	0
Gint	clip_x_origin	GDK_GC_CLIP_X_ORIGIN	gdk_gc_set_clip_origin()	0
Gint	clip_y_origin	GDK_GC_CLIP_Y_ORIGIN	gdk_gc_set_clip_origin()	0
Gint	graphics_exposures	GDK_GC_EXPOSURES	gdk_gc_set_exposures()	TRUE
Gint	line_width	GDK_GC_LINE_WIDTH	gdk_gc_set_line_attributes()	0
GdkLineStyle	line_style	GDK_GC_LINE_STYLE	gdk_gc_set_line_attributes()	GDK_LINE _SOLID
GdkCapStyle	cap_style	GDK_GC_CAP_STYLE	gdk_gc_set_line_attributes()	GDK_CAP _BUTT
GdkJoinStyle	join_style	GDK_GC_JOIN_STYLE	gdk_gc_set_line_attributes()	GDK_JOIN _MITER
Gchar	dash_list[]	none	gdk_gc_set_dashes()	{4, 4}
Gint	dash_offset	None	gdk_gc_set_dashes()	0

16.9 绘图

一旦理解了绘图区、颜色、视件、图形上下文和字体，绘图就会变得很简单了。本节简要介绍了Gdk的相关绘图函数。要注意的是绘图是一种服务器端的操作，例如，如果要画一条线，Xlib会将线的端点传给服务器，服务器用特定的GC(GC也是一种服务器端资源)做实际的绘图操作。创建绘图应用程序时通常要考虑程序的性能。

16.9.1 画点

可以用gdk_draw_point()函数画一个点，或用gdk_draw_points()函数画多个点。点是用当前的前景颜色画的。多个点是由一个GdkPoint数组给出的。GdkPoint结构如下所示：

```
typedef struct _GdkPoint GdkPoint;
struct _GdkPoint
{
    gint16 x;
    gint16 y;
};
```

注意，X坐标是从绘图区的左上角开始的，是一个有符号的16位整数。

函数列表：画点

```
#include <gdk/gdk.h>
void gdk_draw_point(GdkDrawable* drawable,
    GdkGC* gc,
    gint x,
    gint y)

void gdk_draw_points(GdkDrawable* drawable,
    GdkGC* gc,
    GdkPoint* points,
    gint npoints)
```


16.9.2 画线

用`gdk_draw_line()`函数画一条线，将线的端点作为它的参数。要画几条连接着的线，可调用`gdk_draw_lines()`函数，并将点的列表作为参数传到函数中。Gdk会将点连接起来。要画多条不必连接起来的线，可调用`gdk_draw_segments()`函数，将线段列表作为参数传到函数中。

GdkSegment结构是这样定义的：

```
typedef struct _GdkSegment GdkSegment;
struct _GdkSegment
{
    gint16 x1;
    gint16 y1;
    gint16 x2;
    gint16 y2;
};
```

如果在同一个绘画请求中画出的线或者线段的端点是相交的，它们会以 GC中的“连接风格”连接起来。

函数列表：画线

```
#include <gdk/gdk.h>

void gdk_draw_line(GdkDrawable* drawable,
                  GdkGC* gc,
                  gint x1,
                  gint y1,
                  gint x2,
                  gint y2)

void gdk_draw_lines(GdkDrawable* drawable,
                   GdkGC* gc,
                   GdkPoint* points,
                   gint npoints)

void gdk_draw_segments(GdkDrawable* drawable,
                      GdkGC* gc,
                      GdkSegment* segments,
                      gint nsegments)
```

16.9.3 矩形

用`gdk_draw_rectangle()`函数画矩形。其中`filled`参数指明是否填充矩形，设为 TRUE意味着填充它，FALSE则不填充。

函数列表：画矩形

```
#include <gdk/gdk.h>

void gdk_draw_rectangle(GdkDrawable* drawable,
                       GdkGC* gc,
                       gint filled,
                       gint x,
                       gint y,
                       gint width,
                       gint height)
```


16.9.4 画弧

`gdk_draw_arc()`函数画一个椭圆或椭圆弧。弧可以是填充的，也可以是不填充的，第 3 个参数切换填充状态。第 4 到第 7 个参数描绘了一个矩形，椭圆内切于这个矩形；`angle1` 参数是画弧的起始角，它以时钟三点钟位置为 0° ；`angle2` 是绕圆弧旋转的角度，如果值是正的，逆时针旋转，否则顺时针旋转。参数 `angle1` 和 `angle2` 都是以 $1/64$ 度来指定的，所以 360° 就是 360×64 。这样可以更精确地指定弧形的尺寸和形状，而不需使用浮点数。参数 `angle2` 不要超过 360° ，因为对椭圆来说旋转角度超过 360° 没有什么意义。

要画一个圆，在正方形内从 0 到 360×64 画一个弧：

```
gdk_draw_arc(drawable, gc, TRUE,
              0, 0,
              50, 50,
              0, 360*64);
```

要画一个半椭圆，改变长轴和短轴的比例，角度从 0 到 180×64 ：

```
gdk_draw_arc(drawable, gc, TRUE,
              0, 0,
              100, 50,
              0, 180*64);
```

有许多 X 服务器在用来画填充圆弧的边缘时是很难看的，特别是很小的圆看起来会不太圆。要解决这个问题，可以在画圆时把圆的轮廓也画出来。

函数列表：画弧

```
#include <gdk/gdk.h>
void gdk_draw_arc(GdkDrawable* drawable,
                  GdkGC* gc,
                  gint filled,
                  gint x,
                  gint y,
                  gint width,
                  gint height,
                  gint angle1,
                  gint angle2)
```

16.9.5 多边形

`gdk_draw_polygon()` 函数用来画一个填充或不填充的多边形。注意，也可以用 `gdk_draw_lines()` 函数画一个不填充的多边形。这两种方法并没有什么实质区别。`gdk_draw_polygon()` 函数的参数与 `gdk_draw_lines()` 中的参数完全一样。多边形不一定必须是凸多边形。它的边可以是交叉的（自交）。自交的多边形用一种“奇 - 偶规则”来填充，规则是重叠奇数次的多边形区域是不填充的。也就是，如果多边形不是重叠的，它就是完全填充的；如果一个区域重叠了一次，它不会被填充；如果它重叠了两次，它会被填充等等。

函数列表：画多边形

```
#include <gdk/gdk.h>
void gdk_draw_polygon(GdkDrawable* drawable,
                      GdkGC* gc,
                      gint filled,
```

```
GdkPoint* points,  
gint npoints)
```

16.9.6 文本

有两个函数可用于绘制字符串：`gdk_draw_text()`和`gdk_draw_string()`。其中`gdk_draw_text()`是最好的方法。`gdk_draw_text()`用字符串的长度作为参数，而`gdk_draw_string()`函数用`strlen()`函数计算字符串的长度。函数中的`x`和`y`坐标指定要绘制的文本基线的左边位置。文本是用前景颜色绘出的。

在Gdk中没有方法画出缩放的或旋转的文本。`GnomeCanvasText`构件提供了一个较慢的、质量较低的方法来渲染缩放的和旋转的文本。如果需要高质量的按比例缩放的和旋转的文本，则需要使用额外的库函数，比如说对Type 1 fonts 字体使用`ttlib` 库函数，或对True Type fonts 字体使用`FreeType`库函数，或者使用Display Postscript的X扩展(XDPS)。GNU项目组正致力于开发一个免费的XDPS的Linux版本。作为`gnome-print` 库的一部分，Gnome项目组也正在开发一个文本绘图方案。

函数列表：画文本

```
#include <gdk/gdk.h>  
void gdk_draw_string(GdkDrawable* drawable,  
    GdkFont* font,  
    GdkGC* gc,  
    gint x,  
    gint y,  
    const gchar* text)  
  
void gdk_draw_text(GdkDrawable* drawable,  
    GdkFont* font,  
    GdkGC* gc,  
    gint x,  
    gint y,  
    const gchar* text,  
    gint text_length)
```

16.9.7 pixmap像素映射图形

`gdk_draw_pixmap()`从一个像素映射图形中复制一个区域到另一个绘图区（像素映射或窗口）。绘图区的源和目的必须有相同的深度和视件（`visual`）。如果给`width`或`height`参数传递-1值，会将源pixmap图片全部复制过去。源图片可以是任何绘图区，包括窗口，但是如果源是一个窗口，使用`gdk_window_copy_area()`函数将使代码更清晰。下面是`gdk_draw_pixmap()`函数的声明形式。

函数列表：画pixmap图形

```
#include <gdk/gdk.h>  
void gdk_draw_pixmap(GdkDrawable* drawable,  
    GdkGC* gc,  
    GdkDrawable* src,  
    gint xsrc,  
    gint ysrc,
```

```
gint xdest,  
gint ydest,  
gint width,  
gint height)
```

16.9.8 RGB缓冲

Gdk的GdkRGB模块允许将一个图像数据的客户端缓冲复制到一个绘图区。如果要大量操作图像，或将图像数据复制到服务器，最好使用这种方法。因为 pixmap是一种服务器端对象，不能直接操纵 GdkPixmap。用 gdk_draw_point() 函数复制图像数据到服务器中是非常慢的，因为每一个点都要求一个服务器请求（也许还不止一个，因为还需要为每个点更改其 GC）。

本质上来说，GdkRGB用一个称为 GdkImage 的对象在一个请求内将图像数据复制到服务器。这样还是有点慢（大量的数据需要复制），但是 GdkRGB 是已经对此进行了优化，并且如果客户和服务端碰巧是在同一台机器上，它还会使用共享内存。所以在 X 结构中，这是完成这个任务最快的方法。它还会处理一些微妙的问题（比如说适应给定 X 服务器上的颜色表和视件）。

GdkRGB 函数是在一个单独的头文件 gdk/gdkrgb.h 里面定义的。在使用任何 GdkRGB 函数前，必须用 gdk_rgb_init() 函数初始化 GdkRGB 模块，它设置一些要用到的视件和颜色表，以及一些内部的数据结构。

要将 RGB 缓冲复制进去的绘图区必须使用 GdkRGB 的视件和颜色表。如果绘图区是构件的一部分，保证这一点最简单的方法就是在创建构件时将 GdkRGB 视件和颜色表压入构件的视件和颜色表栈中：

```
GtkWidget* widget;  
gtk_widget_push_visual(gdk_rgb_get_visual());  
gtk_widget_push_colormap(gdk_rgb_get_cmap());  
widget = gtk_widget_new();  
gtk_widget_pop_visual();  
gtk_widget_pop_colormap();
```

如果创建包含绘图区的顶级窗口时像上面那么做，而不是在只在创建绘图区时做，当前的 Gtk+ 版本表现还是较好的。不过，原则上可以只对绘图区做这样的工作。

GdkRGB 能理解几种类型的图像数据，包括 24 位和 32 位 RGB 数据、8 位灰度级，以及 8 位的按 RGB 值数组索引的数据（一种客户端 GdkRgbCmap）。本节只介绍最简单的 24 位 RGB 数据，这种缓冲数据是用 gdk_draw_rgb_image() 函数渲染的。有一些单独的函数用于渲染其他缓冲类型，但是所有其他函数本质上的工作原理都是一样的。

24 位 RGB 缓冲是一个一维的字节数组，每个字节三个一组组成像素值（0 位是红色，1 位是绿色，2 位是蓝色）。三个数字描述了数组的大小和字节的位置：

width 是图像每行的像素数（三个字节）。

height 是图像的行数。

Rowstride（行跨距）是行之间的字节数。也就是说，对于一个 rowstride 为 r 的缓冲，如果第 n 行从数组索引 i 开始，那么第 n+1 行从数组索引 i+r 开始。rowstride 不一定是缓冲宽度的三倍；如果源指针和 rowstride 都与 4 字节边界对齐，GdkRGB 会较快。指定一个 rowstride 允许用填充量来达到这个效果。

gdk_rgb_draw_image() 中的 x、y、width 和 height 参数定义了目标可绘区中的一个区域，

RGB缓冲就复制到这个区域。RGB缓冲至少要有width列、height行。RGB缓冲的第0行第0列将复制到可绘区的(x, y)。

gdk_draw_rgb_image 函数中的dither(抖动)参数在有限调色板的显示器上模拟很多颜色。dither只在8位和16位的显示器上管用,24位的显示器的调色板没有什么限制。dither参数是一个枚举类型,有下列几种可能取值:

- GDK_RGB_DITHER_NONE 规定没有抖动。它对画文本或用较少的颜色画线很合适,但是对处理照片不合适。
- GDK_RGB_DITHER_NORMAL 规定在8位的显示器上抖动。但在16位的显示器上不抖动。这通常是质量/性能的一个折衷。
- GDK_RGB_DITHER_MAX 规定在8位和16位的显示器上都抖动。在16位显示器上使用GDK_RGB_DITHER_MAX能够使绘图质量获得提高,但是与性能损失相比可能不值得这么做。

gdk_draw_rgb_image()的gc参数被简单地传递到gdk_draw_image()函数中(GdkRGB在内部使用GdkImage)。最好使用有意义的gc值(比如剪裁屏蔽值、绘图函数,以及subwindow模式)。

函数列表: GdkRGB

```
#include <gdk/gdkrgb.h>
void gdk_rgb_init()
GdkColormap* gdk_rgb_get_cmap()
GdkVisual* gdk_rgb_get_visual()
void gdk_draw_rgb_image(GdkDrawable* drawable,
                        GdkGC* gc,
                        gint x,
                        gint y,
                        gint width,
                        gint height,
                        GdkRGBDither dither,
                        guchar* rgb_buf,
                        gint rowstride)
```


第三部分 Linux GUI生成器Glade

第17章 Glade：GUI生成器

17.1 安装Glade

17.1.1 Glade简介

毫无疑问，使用 Gtk/Gnome 构件编程的概念并不难。然而，使用这些函数存在一些困难：首先是创建程序界面的代码是非常繁琐的，特别是在使用不同的布局构件组装界面元素，创建菜单、工具条等时，不能在编写代码时直接看到显示效果；其次是对代码量较大的程序，可能要将代码放在不同的 C 语言文件中，为它们配置编译选项、写 Makefile 文件也是一项巨大的工程。特别是对于大型项目的开发，这两点尤为突出。应该有一种工具，它可以将我们从这些工作中解放出来，并让我们能够专注于任务的核心。

有没有像 Microsoft Windows 平台下的 Visual Basic、Delphi、C++ Builder 那样快速的开发工具呢？到目前为止，Linux 下还没有功能完备的可视化、快速的编程工具，但是已经有有了一个非常出色的界面生成工具：Glade。它可以用可视化的方法绘制应用程序界面，设置窗口、构件的外观、设置构件信号的回调函数，然后生成 C 语言代码。Glade 目前的版本号是 0.5，也就是说还是发布前版本，不太稳定、成熟。但它的性能已经非常出色。一些专家已经将 Glade 列为今后最有前途的 Linux 快速开发工具。将 Glade、gcc 编译器以及 gdb 结合起来使用，Linux 下的编程将是非常直观的、高效的。

最近，已有一批专家正在致力于开发一个类似于 Visual Basic 的可视化编程工具 gBasic，目的是开发一套可与 VB 媲美的 Basic 编译器。Inprise 公司（以前的 Borland）也可能将要发布 Delphi for Linux。毫无疑问，Linux 今后将成为大部分程序员的重要目标平台，Linux 软件开发将会更加快捷，更加方便。

17.1.2 安装Glade

Glade 是由 Damon Chaplin 创建维护的 Gtk 用户界面生成器。它是基于 GPL 许可的自由软件。只要遵从 GPL 协议，就可以自由地获得其源代码，使用它开发自由软件以及商用的非自由软件。你还可以对它进行修改然后重新发布。所以，如果你对 Glade 有什么意见和建议，或者从 Glade 的源代码中发现了 bug，最好能够和 Glade 的维护者联系，以便于对 Glade 的改进。如果你自己有针对 Glade 的修改意见，也可以将修改方案、代码提交给维护者。

在 <http://glade.pn.org> 网站上能够找到 Glade 最新源代码。一般下载的文件名是 glade-0.5.0.tar.gz。其中的数字代表其版本号。本书出版时，可能会因为版本升级而略有不同。通常将其下载后放在 /usr/src 目录下。

因为安装文件是打包的压缩文件，所以需要先解压缩，然后再编译、安装。安装方法如下：

1) 在shell提示符下输入以下命令，进入文件所在目录：

```
cd /usr/src
```

2) 在shell提示符下输入以下命令，将其解压缩，生成一个归档文件：

```
gunzip glade-0.5.0.tar.gz
```

3) 将tar归档文件展开为目录结构：

```
tar xvf glade-0.5.0.tar
```

这样会将Glade的源代码解压缩到/usr/src/glade-0.5.0目录下。

4) 进入源代码所在目录，运行configure使用程序，配置编译选项，生成 Makefile文件：

```
cd glade-0.5.0
```

```
./configure
```

5) 现在，可以输入make命令编译glade：

```
make
```

依赖于计算机性能，可能需要几分钟到一刻钟时间。编译完成后，将在glade目录下生成Glade可执行文件。

6) 到此为止，已经成功安装了Glade，可以运行了：

```
cd glade
```

```
./glade
```

17.1.3 在Gnome主菜单下为Glade创建菜单项

有时我们可能不想每次都从 xterm下启动Glade，而是希望从 Gnome的主菜单下启动该程序。现在，我们为Glade在Gnome的主菜单上创建一个快捷方式。

1) 选“主菜单/Gnome设置/菜单编辑器”，弹出Gnome菜单编辑器，如图17-1所示。

2) 点击“系统菜单”，然后选“新子菜单”按钮，在Name后输入Glade，在Comment后输入说明性文字“Gtk GUI Builder”，最后点击“保存”按钮。这样，就在Gnome主菜单上创建了一个文件夹。

3) 点击工具条上的“新条目”，在Name后输入Glade，并在Comment后输入说明性文字GUI Builder，在Command后输入Glade可执行文件的路径：

```
/usr/src/glade-0.5.0/glade/glade
```

然后点击“保存”按钮。经过以上步骤，在Gnome的主菜单上就创建了一个快捷方式。



图17-1 Gnome菜单编辑器

17.1.4 在Gnome面板上创建快捷按钮

上面我们为Glade创建了一个快捷方式,就像 Windows 95的“开始/程序”下的快捷方式作用一样。我们还可以将刚才创建的快捷方式放在 Gnome的面板上。

点击“Gnome主菜单/Glade”,打开Glade快捷方式文件夹,按住鼠标左键,将里面的Glade拖到Gnome面板上,然后放下。这样就在 Gnome的面板上创建了一个快捷键。点击它可以直接启动Glade。

注意,本节所展示的菜单编辑器图像是在 TurboLinux 4.0中文版下拷屏得到的。不同的Linux发布版本可能会略有不同,请读者留意。

17.2 用Glade生成图形用户接口

17.2.1 Glade的界面简介

如果已经在Gnome中用前述方法为Glade创建了快捷方式,选择“Gnome主菜单/Glade”下的Glade,即可启动Glade应用程序。Glade的主窗口如图17-2所示:

除主窗口之外,Glade还有构件箱窗口(Palette)、属性编辑器窗口、构件树窗口、剪贴板窗口。下面我们将分别介绍这些窗口的作用。

1. 主窗口

主窗口显示应用程序的最顶层对象,如窗口、弹出菜单、对话框等。要编辑这些对象时,只需在主窗口的列表中双击该对象,即可打开它。选中对象,按 Delete键,就可以将该对象删除。



图17-2 Glade主窗口

主窗口上有文件、编辑、检视、设定、说明,分别说明如下:

(1)“文件”菜单,有以下菜单项:

New Project:创建新的应用程序。因为目前 Glade还不是一个成熟的版本,所以如果有一个应用程序正在使用,它不会提示保存旧文档。这一点一定要留心。

开启旧档:打开已有的应用程序。在Glade中文件是以 glade为后缀保存的,实际上它是一个XML格式的文本文件,描述了界面的构件属性以及其他设置,如构件的信号、回调函数等。

储存文件:保存当前应用程序。一般是以 glade为后缀的XML格式的文本文件。

Build Source Code:Glade的主要目的就是生成创建应用程序界面的代码。所生成的源文件的结构我们将在后面介绍。目前可以生成基于 glib和Gtk+/Gnome构件库的C语言代码,今后还将支持C++和Ada语言。

Project Option:设置应用程序项目的选项,如所用语言、源文件结构等。

结束:退出Glade。

(2)“编辑”菜单,有以下菜单项:

剪下:Glade带一个很有意思的剪贴板,可以将窗口上的构件剪切到剪贴板中。剪贴板只在Glade运行器件时起作用。Glade退出之后,剪贴板中的内容立即消失。剪贴板中可以同时容纳多个构件。最新加到剪贴板的构件是当前构件。

复制:将窗口上的构件复制到剪贴板中。

件。

5. “剪贴板”窗口

在主窗口的“检视”菜单上选择 Show Clipboard，可以打开“剪贴板”窗口。如果有多个构件剪切或复制到剪贴板，可以在此处选择需要的构件，粘贴到窗口上。

图17-4是一个剪贴板窗口示意图，它有三个构件：combo1、label1、button1，其中的combo1是当前构件。当选择“编辑”菜单的“粘贴”菜单项时，会将combo1贴到窗口上。你可以在这里将其他构件设为当前构件。



图17-4 剪贴板窗口

17.2.2 用Glade创建应用程序界面

用Glade能以非常直观的方法生成应用程序界面，类似于 Visual Basic和Delphi。不同点在于Visual Basic和Delphi是一个集成开发环境，不仅可以创建界面，还可以创建完整的应用程序，以及调试、编译等。而 Glade仅仅是一个GUI（图形用户接口）生成器，它只能用于生成创建界面的代码，实现应用程序的功能、编译、调试等工作需要使用其他工具。另外，Gtk+/Gnome的构件与 Visual Basic/Delphi的控件的定位方法有很大的不同，前者使用一种组装技术实现，而后者使用直接定位在窗口上，类似于 GtkFixed构件。

选“文件/New Project”，可以新建一个应用程序。新建程序里面没有任何对象。为程序设计界面要做以下几个工作：创建新窗口，在窗口中将构件定位，为设置构件的属性，为构件的信号设置回调函数。

1. 创建新窗口

在构件箱窗口（Palette）上点击 Gtk+ Basic中的GtkWindow构件，将出现一个，标题为 window1的窗口，这就是应用程序的第一个窗口。选择“检视”（或View）菜单中的 Show Property Editor，可以显示属性编辑器。在其中可以设置窗口的属性，如标题、Border Width、尺寸等，还可以为程序添加新的窗体。另外，使用 GnomeApp构件作为应用程序的主窗口也是一个不错的选择。

2. 在窗口中添加构件

在窗口中添加构件涉及构件的定位方法。可以使用 GtkHBox、GtkVBox、GtkTable等来定位构件，还可以用 GtkFixed定位构件。不过，使用 GtkFixed的潜在问题是如果用户调整窗口尺寸，构件的相对位置和尺寸不会随之变化，窗口可能会看起来很怪。一般情况下，不要使用GtkFixed构件。

(1) 用GtkHBox/GtkVBox定位构件

在构件箱上选 GtkHBox或者GtkVBox，然后在窗口上点击，会弹出一个对话框，询问组装箱应该分成几部分。根据界面规划，选定一个值，然后确定，这时窗口被分为几部分。现在可以在其中放置构件了。要注意的是，其实将GtkHBox(或GtkVBox)划分为几部分并不重要，因为只要需要，随时可以很方便地改变这个值。添加组装箱之后，可以在属性编辑器中设置组装箱的参数：Homogenous、Expand、Fill、Spacing、Padding等。这几个参数的含义请参看前面关于GtkBox的内容。

下一步，在组装箱中添加构件。在构件箱上找到想要的构件，先用鼠标左键点击该构件，

然后在组装盒上的适当位置再点击一次，就可以将这个构件添加到窗口上。你可以依次向窗口添加其他构件。GtkHBox和GtkVBox的组装方法是完全一样的。将这两者结合起来就可以创建非常复杂的界面。

使用组装盒必须遵守 Gtk+/Gnome构件的组装规则。下面是几个技巧（也适用于GtkTable）：

1) 在窗口的某个区域放置一个 GtkFrame构件，然后在 GtkFrame构件中放三个 GtkRadioButton构件。因为 GtkFrame只能容纳一个子构件，需要采取这样的方法：将 GtkFrame定位在窗口上，在 GtkFrame上放一个GtkHBox（分为三部分）作为子构件，然后将 GtkRadioButton构件组装到这个GtkHBox中。

2) 在窗口上放置一个 GtkLabel构件，但是想要让它对鼠标点击响应。因为 GtkLabel没有自己的 X窗口，所以需要使用 GtkEventBox构件：将 GtkEventBox在窗口上定位，并将 GtkLabel添加在GtkEventBox上，然后再设置 GtkEventBox的事件。

3) 在窗口上放一个 GtkList或GtkCList（或其他类似的构件），但是需要这个构件在必要的时候能够自动出现滚动条。因为 GtkList或GtkCList自己并没有滚动条，我们需要使用一个 GtkScrolledWindow为它提供滚动条。实现方法是：在窗口上添加一个 GtkScrolledWindow构件，并在这个滚动窗口构件上添加 GtkList或者GtkCList构件，然后再设置 GtkScrolledWindow构件的显示滚动条的模式——自动或者总是出现（Automatic 或Always）。

4) 在窗口上放一个 GtkNotebook构件，然后在笔记本构件的某一页上放置一个 GtkFrame构件，在 GtkFrame上放三个 GtkRadioButton。注意到 GtkNotebook的每一页都是容器，它的实现方法是：将 GtkNotebook在窗口上定位，然后在笔记本构件的某一页上放一个 GtkFrame，并在 GtkFrame上添加一个 GtkHBox（分为三部分），最后将三个 GtkRadioButton组装到GtkHBox上。

(2) 使用表格构件（GtkTable）定位构件

用鼠标在构件箱上选择 GtkTable构件，在窗口上点击一下，将弹出一个对话框询问表格划分为几行几列，缺省设置是 3×3 。根据规划设定好行数和列数，表格构件就可以添加在窗口上了。实际上以后还可以根据需要随时增加或减少行数和列数。

下一步就是向表格构件中添加构件。在构件箱中选择需要的构件，并在表格上点击，就可以将构件添加到表格上了。注意，即使要让构件占据表格几个单元格，也只能先放在某一个格子上，然后再调整构件的位置。构件在表格上的位置由以下几个参数决定：水平起点坐标、垂直方向起点坐标、跨越的单元格列数、跨越的单元格行数。选中刚添加的构件，在“属性编辑器”窗口上选择 Place标签页，然后设置构件的位置。其中，Cell X对应于构件在表格中的起点坐标，Cell Y指定垂直起点坐标，Col Span指定跨越多少列，Row Span指定跨越多少行。例如，一个 2×2 的表格构件，一个按钮要占据上面两格，它的 Cell X应为0，Cell Y为0，Col Span为2，Row Span为1；另一个按钮占据左下角的单元格，那么它的 Cell X为0，Cell Y为1，Col Span为1，Row Span为1；第三个按钮占据右下角的单元格，那么，它的 Cell X为1，Cell Y为1，Col Span为1，Row Span为1。你也可以将表格和组装盒结合起来使用。

(3) 用GtkFixed定位构件

虽然不鼓励使用 GtkFixed构件，但是在某些情况下使用它还是很方便的，比如在任何情况窗口的大小都不会改变，窗口内的构件也不会改变位置时。

在构件箱上选择 GtkFixed 构件，然后点击窗口，可以将 GtkFixed 添加到窗口上。在构件箱上选择构件，然后在 GtkFixed 构件上点击，构件就会在上面定位。构件在 GtkFixed 上的位置由以下四个参数决定：X、Y、Width、Height。X和Y是构件左上角的坐标，Width和Height是构件的宽度和高度。你可以用鼠标直接拖动构件以调整它的位置，或将鼠标放在构件角部按住拖动以改变它的大小。也可以在属性编辑器的 Place 标签页上设置上面四个参数值。

3. 为窗口添加菜单

为窗口添加菜单是一个很烦琐的工作。Glade 提供了一个很直观的方法来创建菜单。在构件箱上选 GtkMenubar 构件，然后在窗口上点击，为窗口添加一个菜单条。选中菜单条，选择“检视”菜单的 Show Properties Editor，在“属性编辑器”窗口上的 Widget 页，有一个 Edit Menus，点击这个按钮，显示如图 17-5 所示的菜单编辑器。

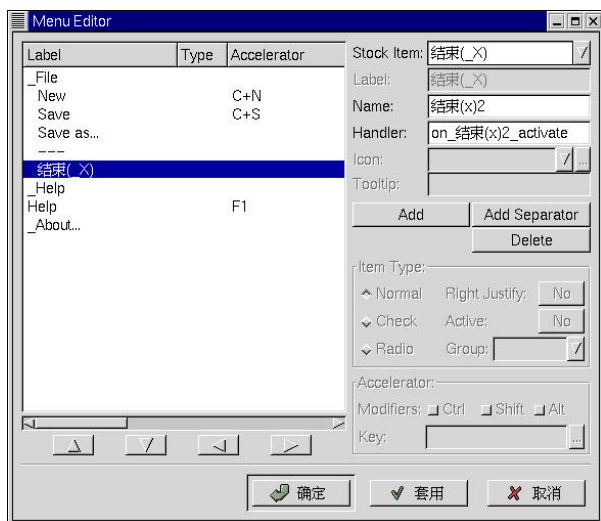


图17-5 Glade菜单编辑器

点击“Add”按钮，添加一个菜单项菜单。在 Label 后填写菜单项的标题，在 Name 后写菜单名称，Handle 后会自动出现处理函数的名称——这也是后面将连接到菜单项上的回调函数。点击向右的箭头可将菜单项设为前面一个菜单项的子菜单。向左的箭头提升一个层次。用向上和向下的箭头可以调整菜单项在菜单中的位置。

可以为菜单设置图片。在 Icon 后的下拉组合框中为菜单项选择一个图片；或者点“Icon”后的按钮，选择一个图片。

选择 Check 无线按钮，可以将菜单项设置成为“检查”菜单项——例如有的程序中的“显示工具条”菜单项，选中表示显示工具条，未选中表示不显示工具条。

选择 Radio 无线按钮，可以将菜单项设置为“无线按钮”类型的菜单项，这时同组的“无线按钮”类菜单项是互斥的——例如有的程序中的对齐方式设置，不可能既是左对齐，又是居中对齐。

在 Accelerator 中可以为菜单项设置快捷键。在 Modifiers 后选中组合键，然后点击 Key 后面的按钮，选择一个按键。建议遵从常用的快捷键设置，比如“新建”菜单项的快捷键一般为 Ctrl+N。

点击 Add Separator，在菜单中间添加一条分隔线。点击 Delete按钮，删除一个菜单项。

注意，不要为带子菜单的菜单项设置快捷键，不要为点击菜单设置图片，也不要将带子菜单的菜单项设置为“检查”或“无线”类型的菜单项。另外，在窗口上添加菜单也要遵守构件组装原则：一般在窗口上放一个 GtkHBox，然后将菜单条组装到这个 GtkHBox中。

做好上面的设置以后，选“确定”按钮，就可以创建一套完整的菜单。

很多程序都提供弹出菜单功能。在窗口上单击鼠标右键，依据具体场合，会弹出一个快捷菜单，给人的感觉是很方便，也很神秘。Glade也提供了一个弹出菜单的实现方法。在构件箱的Gtk Additional上点击Pop Menu，主窗口中添加一个弹出菜单对象。双击它，出现菜单编辑器，与普通的菜单编辑器完全一样。使用上面的步骤创建菜单，Glade会生成一段独立的创建菜单的代码，在需要弹出的场合调用就可以了。

4. 设置构件的属性

上面已经介绍了一部分构件属性。构件的绝大多数属性都可以在属性编辑器中设置，并且可以立即看到效果。

在窗口上选中构件，在属性编辑器中设置它的各项属性。如果在窗口上不能选中它，可以打开“构件树”窗口（选“检视”菜单的 Show Widget Tree菜单项），然后找到该构件，在上面点击右键，从弹出菜单中选 select，再转到属性编辑器，设置它的属性。有一些属性在Glade中还没有完全实现，比如按钮构件的背景颜色和前景颜色、不活动时的颜色等，在Glade中既能够设置属性值，又能够看到效果，但是实际上，生成代码时并没有写到源代码中；编译后，也没有实际效果。今后的 Glade版本可能会实现这些功能。在使用 Glade时对这一点务必注意。

大多数构件在属性编辑器中都有一项 Tooltips。在其中输入说明性的字符串，设计时即可看到它的效果。实际上它为构件添加了一个 GtkTooltip对象。

5. 为构件的信号设置回调函数

在窗口上选择构件，然后选中“属性编辑器”窗口上的Sig页，如图17-6所示。

点击Signal后面的按钮，选择一个信号，然后点击Handler后面的下拉箭头，选择一个回调函数。如果需要，在Data后面输入要传入函数的用户数据，然后点击添加，就为构件的信号设置了一个回调函数。在本例中，为一个名为button1的按钮的clicked信号设置了一个回调函数：on_button1_clicked。目前，Glade不能为构件的子构件的信号设置回调函数。例如，GtkCombo包含一个GtkEntry子构件。如果想对GtkCombo构件combo1的显示文本的变化进行响应，应该为GTK_COMBO(combo1)->entry的changed信号设置回调函数。目前的Glade还不能做到这一点，所以必须在interface.c、callbacks.c以及callbacks.h中手工添加代码。

6. 生成源代码

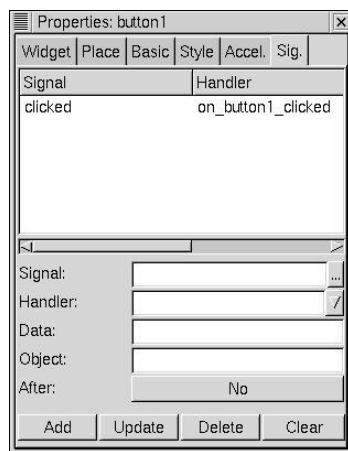


图17-6 为构件的信号设置回调函数

完成上面所列的工作后，实际上应用程序的界面设计已经大体完工了。下一步就是生成源代码，然后实现应用程序的特有功能。

在Glade下，选择“文件”菜单下的 Build Source Code菜单项，或者点击主窗口工具条上的“Build”按钮，可以生成C语言的源代码。这些将是我们应用程序的基础，以后的工作就是通过编码实现程序的各项功能。Glade生成的源代码完全符合Gnome应用程序的编码规范。

一般Glade会创建一个Macros目录（其中包含了编译需要的宏）、一个po目录（用于容纳国际化文件）、一个src目录（源代码）、一个autogen.sh脚本文件，以及其他设置编译选项时要用到的文件。此外还有ChangeLog、Readme、News、Authors等文件（空文件）。应用程序的源代码都放在src目录里。其中包含main.c、interface.c、interface.h、callbacks.c、callbacks.h、support.c、support.h几个主要文件。

Main.c是程序的主文件，它包含了main函数，在它的头部包含了gnome.h文件（包含了gnome.h之后不再需要包含gtk.h、gdk.h、glib.h以及所有用到的构件的头文件）。创建应用程序用户界面的函数都放在interface.c中，interface.h中包含了interface.c中的所有函数声明。所有的回调函数都放在callbacks.c中，callbacks.h文件中包含了callbacks.c中所有函数的声明；support.c文件中包含Glade提供的几个实用函数；support.h包含support.c中的所有函数的声明。

有了上面这些文件之后，下一步就是在这个基础上增加代码以实现应用程序的功能。如果要直接在代码修改界面，可以修改interface.c文件；如果要添加新的回调函数，可以在callbacks.c和interface.c中添加代码。要注意的是，如果增加了新函数，不要忘了在相应的头文件（interface.h、support.h、callbacks.c）里添加函数声明。

另外，support.c中包含了几个由Glade提供的实用函数，它们是：

```
GtkWidget* lookup_widget (GtkWidget *widget,
                           const gchar *widget_name);

GtkWidget* create_pixmap (GtkWidget *widget,
                          const gchar *filename,
                          gboolean     gnome_pixmap);
```

lookup_widget根据提供的构件的名称返回一个构件指针。在代码中调用这个行数来传递指针是非常方便的。create_pixmap用于在interface.c中由文件名创建pixmap图片。

7. 编译用Glade生成的代码

对较大型的程序设置编译选项，以及创建Makefile是很复杂的。一般要联合使用各种GNU工具，如automake、autoconf等，创建一个configure脚本和Makefile.am文件。然后运行configure脚本设置编译选项生成Makefile文件。Glade所生成的C源代码中包含一个名为autogen.sh的脚本。使用它可以轻松完成这一复杂任务。假设用Glade创建了一个应用程序myapp的界面，并已经通过编程实现了所需要的各种功能。源代码存放在/root/myapp下。在shell提示符下执行下面的代码：

```
cd /root/myapp
./autogen.sh
```

autogen.sh脚本会搜索源代码的路径、头文件路径、所需库文件的安装路径，然后生成一个Makefile文件。现在，就可以开始编译了。在shell提示符下输入：

```
make
```

如果源代码没有错误，就会生成所需要的可执行文件了。编译结果一般放在 `src`子目录下。可以尝试运行如下程序：

```
cd src
./myapp
```

如果需要，可以用 `gdb`或者 `xxgdb`调试这个程序。

除了某些功能还没有实现以外，当前 Glade版本中还有一些 bug。最好不要使用构件箱中 Gnome页上的 `GnomeMessageBox`（Gnome消息框），因为它所生成的关于 `GnomeMessageBox` 代码不能正常编译。实际上，创建 `GnomeMessageBox`以及调用它的方法非常简单。

上面仅仅是对 Glade用法的一个简单概述。还有一些 Glade功能这里没有介绍。Glade是非常有希望的GUI生成器，Linux社区的专家对它寄予厚望。可以预见，今后的 Glade功能会更加强大。

第四部分 调试工具

第18章 程序调试

将glib函数库、Gtk+构件库、Gnome库和GCC编译器结合起来可以用来开发非常复杂的应用程序，足以满足绝大多数的商业应用。但是这些还不足以成为一个完整的开发平台。还需要一个高效的调试器，特别是对较大型的应用程序，这一点更为重要。

Linux 包含了一个叫gdb的GNU调试程序。gdb 可以用来调试使C、C++以及Modula - 2语言开发的程序；根据gdb维护者的计划，今后还将支持Fortran语言。gdb是一个强劲的调试器，提供了非常复杂的调试功能。它不仅能够用来调试 GUI应用程序，还可以用来调试非 GUI的程序、守护程序，甚至还可以将 gdb与正在运行的进程连接起来进行调试。可以用 gdb在程序运行时观察程序的内部结构和内存的使用情况。gdb是基于字符的调试器；同时，还有一个图形界面的gdb版本，称为xxgdb。实际上，xxgdb是将gdb做了一个封装，并提供了一个图形接口，内部使用的还是gdb。

gdb是GNU项目的一部分，它是基于 GPL许可协议的。也就是说，只要遵从 GPL协议，就可以自由使用、修改、发布，且不需要为之付费。

下面是gdb和xxgdb 所提供的一些功能：

- 监视程序中变量的值。
- 设置断点以使程序在指定的代码行上停止执行。
- 让程序在指定条件下停止下来，检查程序的运行情况、表达式或变量的变化。
- 可以逐行执行程序代码。
- 运行中改变程序代码，可以直接体验修正 bug后的效果。

这里我们先介绍gdb，然后再介绍xxgdb。

18.1 用gdb调试应用程序

18.1.1 为调试程序做准备

一般大多数Linux的发布版本都包含了gdb。安装时若选择“全部安装”或“安装为开发工作站”，就会安装gdb程序。在shell提示符下输入以下命令：

```
which gdb
```

如果安装了gdb，将会返回gdb的安装路径，一般是 /usr/bin/gdb，否则会什么也不显示。可以在Linux发布版本的光盘上找到gdb的安装文件，一般是gdb-4.18.rpm或者gdb-4.18.tar.gz。安装方法和普通的应用程序的安装方法一样，这里就不做介绍了。

要用gdb调试应用程序，当然首先得有应用程序。所以，要保证编写的应用程序没有语法错误，并且已经调试通过。同时，为了使gdb正常工作，必须使程序在编译时包含调试信息。

调试信息包含了程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。

gdb 利用这些信息使源代码和机器码相关联。

在编译时用 -g 选项打开调试选项。

18.1.2 获得gdb帮助

在命令行上输入 gdb 并按回车键就可以运行 gdb 了, 如果一切正常的话, 将启动 gdb, 可以在屏幕上看到类似的内容:

```
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-pc-linux-gnu".
(gdb)
```

启动gdb后, 可以在命令行上指定很多的选项。输入:

```
help
```

你可以获得 gdb 的帮助信息。如果想要了解某个具体命令 (比如 break) 的帮助信息, 在 gdb 提示符下输入下面的命令:

```
help break
```

屏幕上会显示关于 break 的帮助信息。从返回的信息得知, break 是用于设置断点的命令。另一个获得 gdb 帮助的方法是浏览 gdb 的手册页。在 Linux Shell 提示符下输入:

```
man gdb
```

可以看到 gdb 的手册页。

18.1.3 gdb常用命令

还可以用下面的方式来运行 gdb:

```
gdb filename
```

其中, filename 是要调试的可执行文件。用这种方式运行 gdb 时可以直接指定想要调试的程序。这和启动 gdb 后执行 file filename 命令效果完全一样。你也可以用 gdb 去检查一个因程序异常终止而产生的 core 文件, 或者与一个正在运行的程序相连。

gdb 支持很多的命令, 还可以实现不同的功能。这些命令包含从简单的文件装入到允许检查所调用的堆栈内容的复杂命令。下表列出了在使用 gdb 进行调试时会用到的一些命令。想了解 gdb 的详细使用情况请参考 gdb 的手册页。

file 命令: 装入想要调试的可执行文件。例如:

```
file myapp
```

装入名为 myapp 的应用程序。要注意, 必须指定要装入程序的完全路径, 或者用 cd 命令将工作目录转换到 myapp 所在目录上。

cd 命令: 改变工作目录。例如:

```
cd /root/Projects/myapp
```

将工作目录改变为 /root/Projects/myapp。这和 shell 命令 cd 的作用相同。

pwd命令：返回当前工作目录。例如：

```
pwd
```

将返回当前的工作目录，比如 /root/Projects/myapp。

run命令：执行当前被调试的程序。例如：

```
run
```

开始运行myapp程序。如果设置了断点，会在断点处挂起等待调试。有的应用程序在启动时可以指定选项，或者带参数运行。要调试这种情况，在 run后面加上选项参数表。

kill命令：停止正在调试的应用程序。

list命令：列出正在调试的应用程序的源代码。

break命令：设置断点。例如：

```
break 20
```

表示在第20行设置断点。程序运行到这一行时会挂起，并等待调试。一个应用程序可以设置多个断点。要注意的是，只能在可执行的代码行上设置断点，空行、注释等不可执行的行是不能设置断点的。如果应用程序由多个源程序组成，可以在不同文件中设置断点。例如：

```
break interface.c:30
```

在interface.c中的第30行设置断点。文件名和行号之间是一个冒号（:）。

tbreak命令：设置临时断点。它的语法与 break相同。区别在于用tbreak设置的断点执行一次之后立即消失。

watch命令：设置监视点，监视表达式的变化。例如：

```
watch mytitle
```

当mytitle（假定mytitle是一个变量）的值发生变化时，将应用程序挂起，并显示 mytitle的值。

awatch命令：设置读写监视点，当要监视的表达式被读或写时将应用程序挂起。它的语法与watch命令相同。

rwatch命令：设置读监视点，当监视表达式被读时将程序挂起，等待调试。此命令的语法与watch命令相同。

next命令：执行下一条源代码，但是不进入函数内部。也就是说，将一条函数调用作为一条语句执行。执行这个命令的前提是已经用 run开始了代码的执行。

step命令：执行下一条源代码，进入函数内部。如果调用了某个函数，会跳到函数所在的代码中，等候一步步执行。执行这个命令的前提是已经用 run开始执行代码。

display命令：在应用程序每次停止运行时打印表达式的值。

info break命令：显示当前断点列表，包括每个断点到达的次数。

info files命令：显示调试文件的信息。

info func命令：显示所有的函数名。

info local命令：显示当前函数的所有局部变量的信息。

info prog命令：显示调试程序的执行状态。

print命令：显示表达式的值。

delete命令：删除断点。指定一个断点号码，则删除指定断点。不指定参数则删除所有的断点。

shell命令：执行Linux Shell命令。例如：

```
shell ls
```

可以在不离开gdb的情况下，执行shell命令。

make命令：不退出gdb而重新编译生成可执行文件。

quit命令：退出gdb。

上面列出的命令只是gdb中最常用的一些命令。更多的命令可以在gdb的手册页中找到。

gdb 支持很多与 Linux Shell 程序一样的命令编辑特征。可以像在 bash或tcsh里那样按Tab键让gdb补齐一个唯一的命令，如果不唯一，gdb会列出所有匹配的命令。你还可以用向上和向下的方向键上下翻动历史命令。

18.1.4 gdb 应用举例

下面用一个实例介绍怎样用gdb调试程序。被调试的程序相当简单，但它展示了gdb的典型应用。这也是一般介绍gdb时经常用到的一个例子。

下面列出了将被调试的程序。这个程序称为greeting，它显示一个简单的问候，再用反序将它列出。

```
#include <stdio.h>
main ()
{
    char my_string[] = "hello there";
    my_print (my_string);
    my_print2 (my_string);
}
void my_print (char *string)
{
    printf ("The string is %s\n", string);
}

void my_print2 (char *string)
{
    char *string2;
    int size, i;
    size = strlen (string);
    string2 = (char *) malloc (size + 1);
    for (i = 0; i < size; i++)
        string2[size - i] = string[i];
    string2[size+1] = '\0';
    printf ("The string printed backward is %s\n", string2);
}
```

用gcc编译它：

```
gcc -o test test.c
```

程序执行时显示如下结果：

```
The string is hello there
The string printed backward is
```

输出的第一行是正确的，但第二行打印出的东西并不是我们所期望的。期望的输出应该是：

```
The string printed backward is ereht olleh
```

毫无疑问，my_print2 函数没有正常工作。现在，让我们用 gdb 看看问题究竟出在哪儿，先输入如下命令：

```
gdb greeting
```

如果输入命令时忘了把要调试的程序作为参数传给 gdb，可以在 gdb 提示符下用 file 命令加载它：

```
(gdb) file greeting
```

这个命令加载 greeting 可执行文件，就像在 gdb 命令行里加载它一样。

现在可以用 gdb 的 run 命令来运行 greeting 了。当它运行在 gdb 中时结果大约会像这样：

```
(gdb) run
Starting program: /root/greeting
The string is hello there
The string printed backward is
Program exited with code 041
```

这个输出和在 gdb 外面运行的结果一样。可是，为什么反序打印没有工作呢？为了找出问题所在，我们可以在 my_print2 函数的 for 语句后设一个断点。具体的做法是在 gdb 提示符下执行三次 list 命令，列出源代码：

```
(gdb) list
(gdb) list
(gdb) list
```

每次执行 list 命令会列出 10 行代码。

第一次执行 list 命令的输出如下：

```
1      #include <stdio.h>
2
3      main ()
4      {
5          char my_string[] = "hello there";
6
7          my_print (my_string);
8          my_print2 (my_string);
9      }
10
```

如果按下回车键，gdb 将再执行一次 list 命令，输出下列代码：

```
11     my_print (char *string)
12     {
13         printf ("The string is %s\n", string);
14     }
15
16     my_print2 (char *string)
17     {
18         char *string2;
19         int size, i;
20
```

再按一次回车将列出 greeting 程序的剩余部分：

```
21         size = strlen (string);
```

```
22     string2 = (char *) malloc (size + 1);
23     for (i = 0; i < size; i++)
24         string2[size - i] = string[i];
25     string2[size+1] = '\0'
26     printf ("The string printed backward is %s\n", string2);
27 }
```

根据列出的源程序，可以看到应该将断点设在第 24 行，在 gdb 命令行提示符下输入如下命令设置断点：

```
(gdb) break 24
```

gdb 将做出如下的响应：

```
Breakpoint 1 at 0x139: file greeting.c, line 24
```

```
(gdb)
```

现在再执行 run 命令，将产生如下的输出：

```
Starting program: /root/greeting
```

```
The string is hello there
```

```
Breakpoint 1, my_print2 (string = 0xbfffdc4 "hello there") at greeting.c :24
```

```
24  string2[size-i]=string[i]
```

可以通过设置一个观察 string2[size - i] 变量值的观察点来找出错误的产生原因，做法是键入如下语句：

```
(gdb) watch string2[size - i]
```

gdb 将做出如下响应：

```
Watchpoint 2: string2[size - i]
```

现在可以用 next 命令来一步步的执行 for 循环了：

```
(gdb) next
```

经过第一次循环后，gdb 告诉我们 string2[size - i] 的值是 `h`。gdb 显示如下信息：

```
Watchpoint 2, string2[size - i]
```

```
Old value = 0 '\000
```

```
New value = 104 `h
```

```
my_print2(string = 0xbfffdc4 "hello there") at greeting.c:23
```

```
23 for (i=0; i<size; i++)
```

这个值正是期望的。后来的数次循环的结果也都是正确的。当 i=10 时，表达式 string2[size - i] 的值等于 `e`，size - i 的值等于 1，最后一个字符已经拷贝到新的字符串中了。

如果再把循环执行下去，会看到已经没有值分配给 string2[0] 了，而它是新字符串的第一个字符，因为 malloc 函数在分配内存时把它们初始化为空 (null) 字符。所以 string2 的第一个字符是空字符。这解释了为什么在打印 string2 时没有任何输出。

找出了问题的所在，修正这个错误也就会变得很容易。可以把代码里写入 string2 的第一个字符的偏移量改为 size - 1 而不是 size。这是因为 string2 的大小为 12，但起始偏移量是 0，串内的字符从偏移量 0 到偏移量 10，偏移量 11 为空字符保留。

为了使代码正常工作有很多修改办法。一种是另设一个比字符串的实际大小小 1 的变量。下面是这种解决办法的代码：

```
#include <stdio.h>
main ()
{
```



```
char my_string[] = "hello there";
my_print (my_string);
my_print2 (my_string);
}

my_print (char *string)
{
    printf ("The string is %s\n", string);
}

my_print2 (char *string)
{
    char *string2;
    int size, size2, i;
    size = strlen (string);
    size2 = size - 1;
    string2 = (char *) malloc (size + 1);

    for (i = 0; i < size; i++)
        string2[size2 - i] = string[i];
    string2[size] = '\0';
    printf ("The string printed backward is %s\n", string2);
}
```

18.2 用xxgdb调试应用程序

xxgdb是一个X窗口下的调试器，它实际上是 gdb的一个封装。它的优点是不用记忆复杂的gdb命令，缺点是失去了 gdb的灵活性，且功能也不及 gdb强大。不过，普通应用程序使用xxgdb调试是非常方便的。

要知道是否安装了xxgdb，可在shell提示符下输入以下命令：

```
which xxgdb
```

如果已经安装了xxgdb，会返回xxgdb的路径，一般是/usr/X11R6/bin/xxgdb。

要想获得帮助，只需在shell提示符下输入如下命令：

```
man xxgdb
```

这样就可以获得xxgdb的手册页。你可以从中了解到 xxgdb的界面介绍，以及各个按钮的使用方法等。

在Linux Shell提示符下输入xxgdb，启动xxgdb，界面见图18-1。xxxgdb的界面主要由三个部分组成。最上面的部分是代码窗口，选中应用程序之后，应用程序的源代码将出现在该窗口中。中间部分是命令窗口，命令窗口中有一系列的按钮，对应于 xxgdb中的命令。最下面的部分是显示窗口，用于显示所执行的命令、执行结果和反馈信息。将鼠标放在分界线的黑色区域，可以调整各部分的相对大小。

启动xxgdb时，窗口标题是xxxgdb 1.12，而下面的显示窗口的提示却是 GDB 4.18，从这一点也能够看出xxxgdb实际上只是为gdb提供了一个图形界面。

xxgdb的命令窗口上的按钮根据功能可以大致分为以下几类：执行命令类，以各种方式运行应用程序；断点命令类，用于在程序中设置、取消断点；栈命令类，用于跟踪程序运行中

函数调用栈以及在栈之间移动；数据显示命令类，用于跟踪、显示表达式的值；其他命令类，比如加载文件、搜索、退出 xgdb 等。

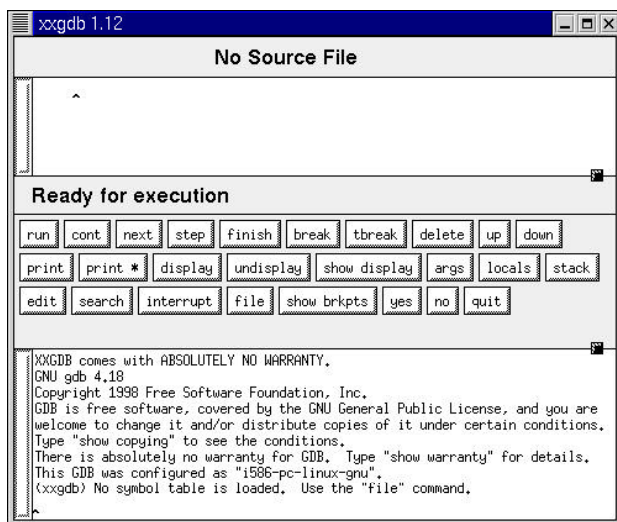


图18-1 用xgdb调试程序

下面简要介绍 xgdb 的几个重要调试功能。

1) 加载文件：点击 file 按钮，弹出一个选择文件对话框，如图 18-2 所示：

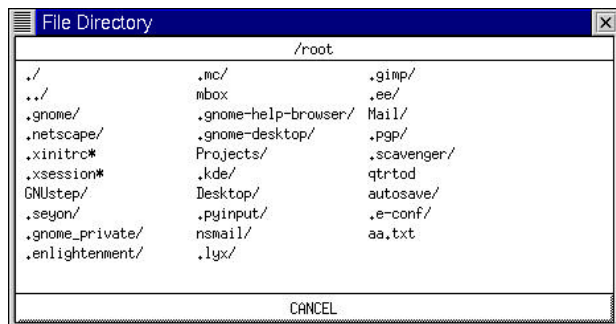


图18-2 在xgdb中选择文件

单击对话框中的目录名，进入应用程序所在目录，然后选择要调试的应用程序的可执行文件。点击 CANCEL 可以取消此次操作。这时程序的源代码会显示在上部的代码窗口中。注意，如果应用程序有多个源文件，这里显示的是包含 main 函数的文件。如果代码较长，代码窗口的左边会显示一个滚动条。用鼠标左键点击滚动条，代码向下滚动；用鼠标右键点击滚动条，代码向上滚动。

2) 加断点：在代码窗口中移动程序代码，找到要添加断点的程序行，在这一行的最前面单击鼠标左键，代码前面会出现一个 “^” 符号，然后，点击 break 按钮，这一行前面会出现一个红色的手形图片，表明这一行已经加上了断点。代码窗口和命令窗口之间会显示断点所在的文件以及行数。如果要设置临时断点（执行一次之后立即取消该断点），应该点击 tbreak 按钮。

如果程序由多个C源程序编译而成，想在其他文件，比如 interface.c中添加断点，需先将相应的文件加载进来。点击 file按钮，选择要跟踪的源文件，然后再在代码窗口中为它设置断点。

3) 显示所有的断点：点击 show brkpts按钮，下部的显示窗口中将显示所有的断点信息，包括断点号（根据设置的先后自动设置）、所在文件以及行号等。

4) 删除断点：在代码窗口选中断点，点击 delete按钮，可以删除该断点。

5) 运行应用程序：点击 run按钮，立即运行程序。如果设置了断点，会在断点处挂起，等待发出其他调试命令。如果没有设置断点，将直接启动应用程序。

6) 继续运行：要从程序停止处继续运行，只需点击 cont按钮（cont实际是continue的简写）。

7) 单步执行：点击 next，继续执行下一行代码，但是不进入函数中；点击 step按钮，继续下一行代码，如果遇到函数调用，会进入函数中。

8) 点击 finish，会继续执行，直到程序返回。例如，正在某个函数中单步执行时，按下 finish会立即执行函数直到函数返回。

9) 点击 stack按钮会显示函数调用的堆栈，选择 up按钮移动调用栈的上一级，选择 down按钮移动调用栈的下一级。

10) 显示表达式的值。在代码中直到找到要监视的表达式，然后双击或按住鼠标左键拖过以选中它，鼠标左键点击 print按钮，将在显示窗口中显示该表达式的值。如果用鼠标右键点击 print按钮，将在一个弹出的数据窗口中显示表达式的值。

11) 显示指针所指向的表达式。在代码窗口中选中要监视的指针，鼠标左键点击 print *按钮，将在显示窗口中显示该指针所指向的对象的信息。如果用鼠标右键点击 print *，将在一个弹出的数据窗口中显示这些信息。

12) 选中一个表达式，点击 display按钮，会在显示窗口中显示表达式的值，且在程序每次挂起时更新一次。

13) 选择第12步中选中的表达式，点击 undisplay，将取消上一步的工作。

14) 点击 search按钮，会弹出一个查找对话框，在里面输入字符串，可以在程序代码中进行搜索。

15) 调试过程中，xxgdb可能会要求用户做出响应，比如对某种情况要求用户回答 y和n。点击 yes按钮向调试器回答 y，点击 no按钮回答 n。

16) 点击 quit按钮将退出调试器。

上面介绍了xxgdb中的较重要的调试功能。用xxgdb调试程序的步骤、方法和gdb中的完全一样，这里就不重复介绍了。

第五部分 附录

附录A GnomeHello源代码

这里是本书中多次用到的 GnomeHello的源代码。它是学习 Gnome编程非常好的材料。源代码共分为五个文件，分别是 hello.c、app.c、app.h、menu.c以及menu.h。

A.1 第一部分：hello.c

```
#include <config.h>
#include <gnome.h>

#include "app.h"

static void session_die(GnomeClient* client, gpointer client_data);

static gint save_session(GnomeClient *client, gint phase,
                        GnomeSaveStyle save_style,
                        gint is_shutdown, GnomeInteractStyle interact_style,
                        gint is_fast, gpointer client_data);

static int greet_mode = FALSE;
static char* message = NULL;
static char* geometry = NULL;

struct poptOption options[] = {
    {
        "greet",
        'g',
        POPT_ARG_NONE,
        &greet_mode,
        0,
        N_("Say hello to specific people listed on the command line"),
        NULL
    },
    {
        "message",
        'm',
        POPT_ARG_STRING,
        &message,
        0,
        N_("Specify a message other than \"Hello, World!\")",
        N_("MESSAGE")
    },
}
```

```
{
    "geometry",
    '\0',
    POPT_ARG_STRING,
    &geometry,
    0,
    N_("Specify the geometry of the main window"),
    N_("GEOMETRY")
},
{
    NULL,
    '\0',
    0,
    NULL,
    0,
    NULL,
    NULL
}
};

int
main(int argc, char* argv[])
{
    GtkWidget* app;

    poptContext pctx;

    char** args;
    int i;

    GSList* greet = NULL;

    GnomeClient* client;

    bindtextdomain(PACKAGE, GNOMELOCALEDIR);
    textdomain(PACKAGE);

    gnome_init_with_popt_table(PACKAGE, VERSION, argc, argv,
                               options, 0, &pctx);

    /* 参数分析*/

    args = poptGetArgs(pctx);

    if (greet_mode && args)
    {
        i = 0;
        while (args[i] != NULL)
        {
            greet = g_slist_prepend(greet, args[i]);
            ++i;
        }
    }
}
```

```

    }
    /* Put them in order */
    greet = g_slist_reverse(greet);
}
else if (greet_mode && args == NULL)
{
    g_error(_("You must specify someone to greet."));
}
else if (args != NULL)
{
    g_error(_("Command line arguments are only allowed with --greet."));
}
else
{
    g_assert(!greet_mode && args == NULL);
}

poptFreeContext(pctx);

/* 会话管理 */

client = gnome_master_client ();
gtk_signal_connect (GTK_OBJECT (client), "save_yourself",
                    GTK_SIGNAL_FUNC (save_session), argv[0]);
gtk_signal_connect (GTK_OBJECT (client), "die",
                    GTK_SIGNAL_FUNC (session_die), NULL);

app = hello_app_new(message, geometry, greet);

g_slist_free(greet);

gtk_widget_show_all(app);

gtk_main();

return 0;
}

static gint
save_session (GnomeClient *client, gint phase, GnomeSaveStyle save_style,
              gint is_shutdown, GnomeInteractStyle interact_style,
              gint is_fast, gpointer client_data)
{
    gchar** argv;
    guint argc;

    argv = g_malloc0(sizeof(gchar*)*4);
    argc = 1;

    argv[0] = client_data;

```



```
if (message)
{
    argv[1] = "--message";
    argv[2] = message;
    argc = 3;
}

gnome_client_set_clone_command (client, argc, argv);
gnome_client_set_restart_command (client, argc, argv);

return TRUE;
}

static void
session_die(GnomeClient* client, gpointer client_data)
{
    gtk_main_quit ();
}
```

A.2 第二部分 : app.h

```
#ifndef GNOMEHELLO_APP_H
#define GNOMEHELLO_APP_H

#include <gnome.h>

GtkWidget* hello_app_new(const gchar* message,
                          const gchar* geometry,
                          GSList* greet);

void        hello_app_close(GtkWidget* app);

#endif
```

A.3 第三部分 : app.c

```
#include <config.h>
#include "app.h"
#include "menus.h"

/* 保留一个所有打开的窗口的链表 */
static GSList* app_list = NULL;

static gint delete_event_cb(GtkWidget* w, GdkEventAny* e, gpointer data);
static void button_click_cb(GtkWidget* w, gpointer data);

GtkWidget*
hello_app_new(const gchar* message,
              const gchar* geometry,
              GSList* greet)
{
```

```
GtkWidget* app;
GtkWidget* button;
GtkWidget* label;
GtkWidget* status;
GtkWidget* frame;

app = gnome_app_new(PACKAGE, _("Gnome Hello"));

frame = gtk_frame_new(NULL);

button = gtk_button_new();

label = gtk_label_new(message ? message : _("Hello, World!"));

gtk_window_set_policy(GTK_WINDOW(app), FALSE, TRUE, FALSE);
gtk_window_set_default_size(GTK_WINDOW(app), 250, 350);
gtk_window_set_wmclass(GTK_WINDOW(app), "hello", "GnomeHello");

gtk_frame_set_shadow_type(GTK_FRAME(frame), GTK_SHADOW_IN);

gtk_container_set_border_width(GTK_CONTAINER(button), 10);

gtk_container_add(GTK_CONTAINER(button), label);

gtk_container_add(GTK_CONTAINER(frame), button);

gnome_app_set_contents(GNOME_APP(app), frame);

status = gnome_appbar_new(FALSE, TRUE, GNOME_PREFERENCES_NEVER);

gnome_app_set_statusbar(GNOME_APP(app), status);

hello_install_menus_and_toolbar(app);

gtk_signal_connect(GTK_OBJECT(app),
                  "delete_event",
                  GTK_SIGNAL_FUNC(delete_event_cb),
                  NULL);

gtk_signal_connect(GTK_OBJECT(button),
                  "clicked",
                  GTK_SIGNAL_FUNC(button_click_cb),
                  label);

if (geometry != NULL)
{
    gint x, y, w, h;
    if ( gnome_parse_geometry( geometry,
                              &x, &y, &w, &h ) )
    {
        if (x != -1)
```

```
{
    gtk_widget_set_uposition(app, x, y);
}

if (w != -1)
{
    gtk_window_set_default_size(GTK_WINDOW(app), w, h);
}
}
else
{
    g_error(_("Could not parse geometry string `%s`"), geometry);
}
}

if (greet != NULL)
{
    GtkWidget* dialog;
    gchar* greetings = g_strdup(_("Special Greetings to:\n"));
    GSList* tmp = greet;

    while (tmp != NULL)
    {
        gchar* old = greetings;

        greetings = g_strconcat(old,
                                (gchar*) tmp->data,
                                "\n",
                                NULL);

        g_free(old);

        tmp = g_slist_next(tmp);
    }

    dialog = gnome_ok_dialog(greetings);

    g_free(greetings);

    gnome_dialog_set_parent(GNOME_DIALOG(dialog), GTK_WINDOW(app));
}

app_list = g_slist_prepend(app_list, app);

return app;
}

void
hello_app_close(GtkWidget* app)
{

```

```
g_return_if_fail(GNOME_IS_APP(app));

app_list = g_slist_remove(app_list, app);

gtk_widget_destroy(app);

if (app_list == NULL)
{
    /* No windows remaining */
    gtk_main_quit();
}

static gint
delete_event_cb(GtkWidget* window, GdkEventAny* e, gpointer data)
{
    hello_app_close(window);

    /* 阻止销毁窗口，因为我们在hello_app_close中销毁它
    */
    return TRUE;
}

static void
button_click_cb(GtkWidget* w, gpointer data)
{
    GtkWidget* label;
    gchar* text;
    gchar* tmp;

    label = GTK_WIDGET(data);

    gtk_label_get(GTK_LABEL(label), &text);

    tmp = g_strdup(text);

    g_strreverse(tmp);

    gtk_label_set_text(GTK_LABEL(label), tmp);

    g_free(tmp);
}
```

A.4 第四部分：menus.h

```
#ifndef GNOMEHELLO_MENUS_H
#define GNOMEHELLO_MENUS_H

#include <gnome.h>

void hello_install_menus_and_toolbar(GtkWidget* app);
```

```
#endif
```

A.5 第五部分 : menus.c

```
#include <config.h>
#include "menus.h"
#include "app.h"

static void nothing_cb(GtkWidget* widget, gpointer data);
static void new_app_cb(GtkWidget* widget, gpointer data);
static void close_cb (GtkWidget* widget, gpointer data);
static void exit_cb (GtkWidget* widget, gpointer data);
static void about_cb (GtkWidget* widget, gpointer data);

static GnomeUIInfo file_menu [] = {
    GNOMEUIINFO_MENU_NEW_ITEM(N_("New Hello"),
                               N_("Create a new hello"),
                               new_app_cb, NULL),

    GNOMEUIINFO_MENU_OPEN_ITEM(nothing_cb, NULL),

    GNOMEUIINFO_MENU_SAVE_ITEM(nothing_cb, NULL),

    GNOMEUIINFO_MENU_SAVE_AS_ITEM(nothing_cb, NULL),

    GNOMEUIINFO_SEPARATOR,

    GNOMEUIINFO_MENU_CLOSE_ITEM(close_cb, NULL),

    GNOMEUIINFO_MENU_EXIT_ITEM(exit_cb, NULL),

    GNOMEUIINFO_END
};

static GnomeUIInfo edit_menu [] = {
    GNOMEUIINFO_MENU_CUT_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_COPY_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_PASTE_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_SELECT_ALL_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_CLEAR_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_UNDO_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_REDO_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_FIND_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_FIND_AGAIN_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_REPLACE_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_MENU_PROPERTIES_ITEM(nothing_cb, NULL),
    GNOMEUIINFO_END
};

static GnomeUIInfo help_menu [] = {
```

```
    GNOMEUIINFO_HELP ("gnome-hello"),

    GNOMEUIINFO_MENU_ABOUT_ITEM(about_cb, NULL),

    GNOMEUIINFO_END
};

static GnomeUIInfo menu [] = {
    GNOMEUIINFO_MENU_FILE_TREE(file_menu),
    GNOMEUIINFO_MENU_EDIT_TREE(edit_menu),
    GNOMEUIINFO_MENU_HELP_TREE(help_menu),
    GNOMEUIINFO_END
};

static GnomeUIInfo toolbar [] = {
    GNOMEUIINFO_ITEM_STOCK (N_("New"),
                            N_("Create a new hello"),
                            nothing_cb, GNOME_STOCK_PIXMAP_NEW),

    GNOMEUIINFO_SEPARATOR,

    GNOMEUIINFO_ITEM_STOCK (N_("Prev"),
                            N_("Previous hello"),
                            nothing_cb, GNOME_STOCK_PIXMAP_BACK),
    GNOMEUIINFO_ITEM_STOCK (N_("Next"),
                            N_("Next hello"),
                            nothing_cb, GNOME_STOCK_PIXMAP_FORWARD),

    GNOMEUIINFO_END
};

void
hello_install_menus_and_toolbar(GtkWidget* app)
{
    gnome_app_create_toolbar_with_data(GNOME_APP(app), toolbar, app);
    gnome_app_create_menus_with_data(GNOME_APP(app), menu, app);
    gnome_app_install_menu_hints(GNOME_APP(app), menu);
}

static void
nothing_cb(GtkWidget* widget, gpointer data)
{
    GtkWidget* dialog;
    GtkWidget* app;

    app = (GtkWidget*) data;

    dialog = gnome_ok_dialog_parented(
        _("This does nothing; it is only a demonstration."),
        GTK_WINDOW(app));
}
```

```
}

static void
new_app_cb(GtkWidget* widget, gpointer data)
{
    GtkWidget* app;

    app = hello_app_new(_("Hello, World!"), NULL, NULL);

    gtk_widget_show_all(app);
}

static void
close_cb(GtkWidget* widget, gpointer data)
{
    GtkWidget* app;

    app = (GtkWidget*) data;

    hello_app_close(app);
}

static void
exit_cb(GtkWidget* widget, gpointer data)
{
    gtk_main_quit();
}

static void
about_cb(GtkWidget* widget, gpointer data)
{
    static GtkWidget* dialog = NULL;
    GtkWidget* app;

    app = (GtkWidget*) data;

    if (dialog != NULL)
    {
        g_assert(GTK_WIDGET_REALIZED(dialog));
        gdk_window_show(dialog->window);
        gdk_window_raise(dialog->window);
    }
    else
    {
        const gchar *authors[] = {
            "Havoc Pennington <hp@pobox.com>",
            NULL
        };

        gchar* logo = gnome_pixmap_file("gnome-hello-logo.png");
```

```
dialog = gnome_about_new (_("GnomeHello"), VERSION,  
                           "(C) 1999 Havoc Pennington",  
                           authors,  
                           _("A sample GNOME application."),  
                           logo);  
  
g_free(logo);  
  
gtk_signal_connect(GTK_OBJECT(dialog),  
                  "destroy",  
                  GTK_SIGNAL_FUNC(gtk_widget_destroyed),  
                  &dialog);  
  
gnome_dialog_set_parent(GNOME_DIALOG(dialog), GTK_WINDOW(app));  
  
gtk_widget_show(dialog);  
}  
}
```


附录B 在线资源

B.1 获得Gtk+/Gnome库

因为不同发布版本的编译指令可能是不同的。最好从 <http://www.gtk.org/> 和 <http://www.gnome.org/> 获得Gtk+和Gnome的最新稳定发布版本，然后根据源代码提供的编译指令编译、安装。

B.2 网站

Gtk+和Gnome都有自己的网站：<http://www.gtk.org/> 和<http://www.gnome.org/>。

在Gtk+站点 (<http://www.gtk.org/>)，注意以下页面：

- <http://www.gtk.org/rdp/> 是Gtk+参考文档项目 (Reference Documentation Project)，在这里可以找到Gtk+ API的参考材料。
- <http://www.gtk.org/faq/> 有Gtk+的FAQ，最好不要在邮件列表里面问这里已有答案的问题。

在Gnome站点 (<http://www.gnome.org/>)，注意以下页面：

- <http://www.gnome.org/lxr/> 包含了Gtk+/Gnome CVS 服务器中所有代码的可浏览、搜索的超文本版本。它包含Gtk+和Gnome，还有许多应用程序代码。
- <http://bugs.gnome.org/> 允许你浏览Gtk+和Gnome的bug报告，还可以递交新的bug。如果发现bug，请在此处提交报告，以便维护人员能跟踪它。
- <http://developer.gnome.org> 包含针对Gnome开发人员的非常全面的示例代码。

下面几个网站也很不错：

<http://www.linuxdev.net>：针对Linux开发人员的站点。

<http://www.linuxaid.com.cn>：中文站点，有很多中文版的文档。

<http://www.turbolinux.com.cn>：拓林思公司 (TurboLinux) 的中文网站。

B.3 邮件列表

下面是一些包含Gtk+和Gnome开发主题的邮件列表：

- gtk-list@redhat.com 对使用Gtk+、阅读Gtk+源代码、讨论Gtk+的错误方面的问题是非常合适的。
- gnome-list@gnome.org 流量非常高，它是通用的Gnome邮件列表，讨论所有与Gnome相关的话题。
- gnome-devel-list@gnome.org 如果了解Gnome的库以及怎样用它们写程序，这个邮件列表是非常好的。它对讨论库开发、或者提交修改库的补丁等也很合适。
- gtk-app-devel-list@redhat.com 是另一个gtk邮件列表，主要针对应用程序开发。大多数适用于这个邮件列表的问题也适用于 gtk-list@redhat.com。但是，如果你在意的话，

app-devel-list的流量较少。

- gtk-devel-list@redhat.com 主要讨论开发 glib、Gdk 以及 Gtk+ 库。它不讨论用这些库开发应用程序。
- gnome-announce-list@gnome.org 发布一些关于 Gnome 和 Gnome 上的应用程序的公告。邮件量较少。

要订阅上面的邮件列表，在邮件列表的名字后面加一个“-request”作为收信地址，给这个地址发一封电子邮件，主题为“subscribe my-address@wherever.net”。例如，我想订阅 gtk-list@redhat.com，可以给 gtk-list-request@redhat.com 发一个邮件，主题为“subscribe hp@pobox.com”。

可以在邮件列表上提出关于 Gtk+ 和 Gnome 的问题。除非你知道他们对某些特别的代码具有特别的知识，尽量不要私下与开发员通信。实际上，总能够从邮件列表中得到更快、更好的回应。

附录C Gtk+/Gnome对象总览

本书所介绍的对象只是 Gtk+/Gnome对象中的一部分。还有很多有趣的内容因为篇幅的原因没有涉及。同时，Gtk+/Gnome也是在不断发展的。有一些构件现在使用得很广泛，也许今后版本会推出一个功能更多，更稳定的新构件。有些构件现在还是实验性的，今后也许会变成正式的构件。

本附录是Gtk+和Gnome对象层次的快速教程。它包含 Gtk+和Gnome库中的GtkObject对象和它所有的子类，每个对象有一个简要的描述以及该对象的头文件。这些有助于为特定的任务选用合适的构件。

每个对象都列出了它的头文件。不过在程序中包含 `gtk.h` 就可以包含所有的 Gtk+ 头文件，使用 `gnome.h` 头文件就可以包含所有的其他头文件。

一些对象被描述为“抽象基类”意指只有该对象的子类才能实例化，但是所有的子类都可以由它的基类接口来操纵。

作为通行的规则，应该尽可能使用最有针对性的对象。也就是，可以用一个 `GtkWindow` 作为应用程序的主窗口，但是 `GnomeApp` 才是更好的选择。可以用 `GnomeDialog` 做一个“关于”对话框，但是最好还是使用 `GnomeAbout`。这能够最大程度保证用户界面的一致性，也可以省一些事。

Gtk+和Gnome都带一个“test”程序，分别称为 `testgtk` 和 `test-gnome`。这些程序用于测试每个库中的构件和其他的特性。它们也是优秀的示例代码来源，同时还是一个浏览可用构件并选用合适构件的好方法。

C.1 GtkObject

库：Gtk+

头文件：`gtk/gtkobject.h`

描述：GtkObject是Gtk+的对象层次的基础。它不是一个图形化的组件。它实现了引用数、连接键/值对到对象上，以及对象解构(按C++术语，“虚解构的函数”)等接口。GtkObject本身在Gtk+对象系统中起着很重要的作用。Gtk+的信号/回调函数的基础结构是工作在GtkObject对象上的，也就是，信号是由特定的GtkObject对象引发的，回调函数连接到特定的对象和信号。

C.2 构件

构件也是Gtk+之所以存在的理由。构件是 `GtkWidget` 的子类，`GtkWidget` 是 `GtkObject` 的子类。一个构件代表了屏幕上的一个矩形区域，它也许是纯粹装饰性的，交互式的控件，或者是一个控制子构件排列的容器。

1. GtkWidget

库：Gtk+

头文件：gtk/gtkwidget.h

描述：GtkWidget是所有构件的父类。GtkWidget是一个抽象基类。

2. GtkContainer

库：Gtk+

头文件：gtk/gtkcontainer.h

描述：是能包含其他构件的抽象基类。

3. GtkBin

库：Gtk+

头文件：gtk/gtkbin.h

描述：GtkBin是只能包含一个子构件的容器的抽象基类。它提供了 GtkWidget接口的缺省实现方法，所以为它创建子类是很容易的。

4. GtkWindow

库：Gtk+

头文件：gtk/gtkwindow.h

描述：GtkWindow代表一个顶级对话框和应用程序窗口。作为 Gtk+中的主要顶级构件，它有很多特殊的职责，例如，它维护了当前的键盘焦点并决定它自己的尺寸分配（而不是从它的父构件接收）。典型情况下，Gnome应用程序用GnomeApp构件作为应用程序的主窗口，充分利用它的附加功能。对对话框，在Gnome应用程序中应该使用GnomeDialog，对Gtk+应用程序使用GtkDialog构件。当然还有几个特制的对话框子类可以使用。如果没有合适的GtkWindow子类，可以直接使用GtkWindow窗口。

警告：如果接收到“delete_event”信号，GtkWindow会被自动销毁。要防止这种情形发生，必须设置一个新的要运行的信号处理程序，并且信号处理程序必须返回TRUE。这是很常见的Gtk+程序错误。GnomeDialog构件将有助于处理这种情况。

让GtkWindow作为最后调用gtk_widget_show()显示的构件是一个好主意。大多数构件直到它们的父构件容器映射到屏幕(放到屏幕上)才会实际映射到屏幕上。但是GtkWindow没有父构件，它会立即出现在屏幕上。所以如果在窗口显示之后再显示它的子构件，将会看到屏幕闪烁。

5. GnomeDialog

库：Gnome

头文件：libgnomeui/gnome-dialog.h

描述：在Gnome应用程序中所有的对话框都应该使用GnomeDialog(或其子类)构件。如果没有使用Gnome，GnomeDialog构件还是很有用的，因为它真正实现了一个对话框必须有的所有基本特性。强烈建议研究GnomeDialog的源代码(如果应用程序是基于GPL的，甚至还可以剪切和粘贴其中的代码)。

6. GnomeAbout

库：Gnome

头文件：libgnomeui/gnome-about.h

描述：GnomeAbout是一个“关于...”对话框，与Gnome的about对话框外观是一致的。

7. GnomeMessageBox

库：Gnome

头文件：libgnomeui/gnome-messagebox.h

描述：GnomeMessageBox是一个简单的GnomeDialog，里面预先组装了一个标签和一个小图标。图标对应于一个“信息框类型”，比如一个警告消息、一个错误、一个问题。这个图标让用户快速判定对话框的目的。

8. GnomePropertyBox

库：Gnome

头文件：libgnomeui/gnome-propertybox.h

描述：GnomePropertyBox是一个对话框，可以在程序中用于设置应用程序参数选择值或一些用户可见对象的属性值。它有“apply”、“OK”、“Close”以及“Help”几个按钮，“OK”按钮等价于按“Apply”后再按下“Close”按钮。

9. GnomeScores

库：Gnome

头文件：libgnomeui/gnome-scores.h

描述：GnomeScores用于跟踪和显示所有高分表。大多数Gnome游戏都使用这个构件。

10. GnomeApp

库：Gnome

头文件：libgnomeui/gnome-app.h

描述：GnomeApp是一个专门用作应用程序主窗口的GtkWidget子类。它为可选的工具条、菜单条和状态条留有空间。应用程序的“文档”放在构件中央的特别区域。

11. GtkDialog

库：Gtk+

头文件：gtk/gtkdialog.h

描述：GtkDialog是一个GtkWindow窗口，带有三个预置的构件：一个GtkVBox用于显示对话框内容，一个GtkSeparator，以及一个GtkHBox用于放置对话框的按钮。GtkDialog用处不大，所有的Gnome程序都应该使用GnomeDialog。

12. GnomeFontSelector

库：Gnome

头文件：libgnomeui/gnome-font-selector.h

描述：GnomeFontSelector是一个已经过时的字体选择对话框，现已被GtkFontSelectionDialog构件取代。GtkFontSelectionDialog包含一个GtkFontSelection。Gnome程序应该在GnomeDialog中放一个GtkFontSelection构件作为字体选择对话框，因为GtkFontSelectionDialog并不使用GnomeDialog，因而它的外观与Gnome风格有所不同。不应该使用GnomeFontSelector。

13. GtkInputDialog

库：Gtk+

头文件：gtk/gtkinputdialog.h

描述：GtkInputDialog是一个对话框，用于选择和设置使用X输入扩展的设备（例如绘图板）。它先于Gnome写成，没有使用GnomeDialog，所以在Gnome应用程序中显得有点滑稽，但是如果不重写该构件，又不用这个构件，就没有合适的工具。

14. GtkColorSelectionDialog

库：Gtk+

头文件：gtk/gtkcolorsel.h

描述：GtkColorSelectionDialog是一个对话框，包含一个 GtkColorSelection。Gnome应用程序应该在 GnomeDialog 对话框中手工放置一个 GtkColorSelection，或使用 GnomeColorPicker。

15. GtkFileSelection

库：Gtk+

头文件：gtk/gtkfilesel.h

描述：GtkFileSelection是一个文件选择对话框(与大多数其他以“selection”结尾的构件不一样，其他组合构件意味着放在一个对话框内)。不幸的是，对这个构件还没有 Gnome 替代品，所以 Gnome 应用程序使用它，尽管它的外观有点不太连贯。

16. GtkFontSelectionDialog

库：Gtk+

头文件：gtk/gtkfontsel.h

描述：GtkFontSelectionDialog是一个对话框，包含一个 GtkFontSelection。Gnome 应用程序应该用 包含一个 GtkFontSelection 的 GnomeDialog 对话框构件，而不是用这个构件。

17. GtkPlug

库：Gtk+

头文件：gtk/gtkplug.h

描述：GtkPlug是一个顶级窗口，可以嵌入到一个在其他应用程序中运行的 GtkSocket 构件中。换句话说，GtkSocket 构件是进程中的一个“洞”，能包含一个可以运行其他程序的 GtkPlug 构件。

18. GtkButton

库：Gtk+

头文件：gtk/gtkbutton.h

描述：GtkButton是一个简单的矩形按钮。它是能包含一个子构件的容器；通常它包含一段文本或像素映射图片，但是它能包含所有构件。

19. GnomeColorPicker

库：Gnome

头文件：libgnomeui/gnome-color-picker.h

描述：GnomeColorPicker是一个按钮，包含一个小彩色方块指示当前选中的颜色。当点击按钮时，它创建一个颜色选择对话框，用来改变颜色。

20. GnomeFontPicker

库：Gnome

头文件：libgnomeui/gnome-font-picker.h

描述：GnomeFontPicker与GnomeColorPicker类似，它是一个按钮，上面显示当前选中的字体，当点击它时，将弹出一个字体对话框。

21. GnomeHRef

库：Gnome

头文件：libgnomeui/gnome-href.h

描述：GnomeHRef是一个无边框的按钮，在上面显示一个超级链接。点击用户时，Gnome指向用户的浏览器，连接到超级链接的目标 URL，或者启动一个新的浏览器实例。用于连接到指定URL的命令可以在Gnome控制中心配置。

22. GtkToggleButton

库：Gtk+

头文件：gtk/gtktogglebutton.h

描述：GtkToggleButton看起来更像一个正常的 GtkButton按钮。然而，它倾向于表现一个可切换的状态。当按钮是“活动”的时，按钮是按下去的。通常应该使用 GtkCheckButton而不是GtkToggleButton，Gtkcheck Button经常比GtkToggleButton看起来更好，并且能给用户一个更好的概念：按钮代表一种可切换状态。

23. GtkCheckButton

库：Gtk+

头文件：gtk/gtkcheckbutton.h

描述：GtkCheckButton和GtkToggleButton差不多，但是外观不一样(它看起来像一个左边有小按钮的标签)。在多数情况，GtkcheckButton是比GtkToggleButton更好的选择，因为切换按钮不能给用户一个视觉上的提示表明它是可切换状态。

24. GtkRadioButton

库：Gtk+

头文件：gtk/gtkradiobutton.h

描述：GtkRadioButton代表几个互斥选项中的一个。GtkRadioButton是放在“组”里的。在一个组里，只有一个按钮可以是活动的。GtkOptionMenu也可以用于代表互斥的选项。如果选项较多，选项菜单可能是更好的选择。如果有大量的选项，GtkList或GtkCList也许更合适。

25. GtkOptionMenu

库：Gtk+

头文件：gtk/gtkoptionmenu.h

描述：GtkOptionMenu从许多选项中显示当前的活动选项，点击它时，将弹出一个菜单，允许用户设置新的活动选项。选项菜单构件还有点问题；当请求尺寸时，它不会考虑菜单项的尺寸。如果在选项菜单中包含标签，标签经常会被截短。最好的方法就是通过对它的父构件容器设置合适的选项，给它指定比实际需要更多的空间。

26. GnomeDockItem

库：Gnome

头文件：libgnomeui/gnome-dock-item.h

描述：GnomeDockItem是一个容器，让它的子构件显示在一个 GnomeDock上。Dock items 能够与它的父窗口分开，并放在桌面的任何地方。它们还可以在 dock内移动。GnomeDock可以让用户重新排列工具条。GnomeDockItem提供了一个“手柄”，用于拖动它的子构件。GnomeApp构件在内部使用GnomeDock构件，这样，Gnome工具条可以重新定位。

27. GtkAlignment

库：Gtk+

头文件：gtk/gtkalignment.h

描述：GtkAlignment是一种不可见容器，用于在空间内对齐子构件。要设置两个因子，每个都要X和Y方向的值；对齐值0.0是左对齐(或顶部对齐)并且1.0是右对齐(或底部对齐)。对齐值为0.5时，子构件在对齐方向居中。比例因子决定子构件应该怎样填充它并不需要的额外空间。如果是0.0，子构件只占据它请求的值。如果是1.0，构件将占据所有的可用空间。（实际上，比例因子1.0 scale使alignment因子不起作用。）

28. GtkFrame

库：Gtk+

头文件：gtk/gtkframe.h

描述：框架构件在它的子构件周围画一个装饰性的框。它还可以有一个标题描述这个框中的内容。要关闭标题，将它设置为 NULL。

29. GtkAspectFrame

库：Gtk+

头文件：gtk/gtkaspectframe.h

描述：GtkAspectFrame用于控制它的子构件的纵横比。与 GtkAlignment相似，它也允许将子构件在两个方向对齐。可以指定一个纵横比，或要求保留子构件的尺寸请求的比率。视觉外观上GtkAspectFrame与GtkFrame完全一样。

30. GtkItem

库：Gtk+

头文件：gtk/gtkitem.h

描述：GtkItem是一个用于列表项、树数据项、菜单项的抽象基类。这种数据项是可以“选中”、“取消选择”和“切换”的构件。

31. GtkMenuItem

库：Gtk+

头文件：gtk/gtkmenuItem.h

描述：菜单项是一个不可见的容器，它是唯一一种能够显示在 GtkMenu上的构件。典型情况下，可以在菜单项上放置一个标签，或者一个标签加一个像素映射图片以指明它的功能。如果菜单项没有子构件，它将绘出一个分隔线。它节省了在菜单项上添加 GtkSeparator 构件的额外开销。

32. GtkCheckMenuItem

库：Gtk+

头文件：gtk/gtkcheckmenuItem.h

描述：GtkCheckMenuItem是一种菜单项，功能与 GtkCheckButton差不多，在它的子构件的左边有一个小按钮，它可以是“活动”的或者是“不活动”的。因为它是 GtkMenuItem的子类，它也可以出现在菜单中。

33. GtkRadioMenuItem

库：Gtk+

头文件：gtk/gtkradiomenuitem.h

描述：GtkRadioMenuItem与GtkRadioButton类似，它允许用户从一系列互斥选项中做出选择。因为它是GtkMenuItem的子类，它可以出现在菜单上。

34. GtkPixmapMenuItem

库：Gnome

头文件：libgnomeui/gtkpixmapmenuitem.h

描述：GtkPixmapMenuItem实际上是一个Gnome构件，尽管它的名字带“Gtk”。这个构件有一个非常特别的问题：当菜单包含检查菜单或无线菜单项时，Gtk会将所有菜单项的子构件缩进，为检查菜单和无线菜单项留出空间。Gnome在菜单旁边放一个像素映射图片；GtkPixmapMenuItem将图片缩进，与Gtk+缩进检查菜单和无线菜单一样。如果只添加一个图片和一个标签到GtkMenuItem中，与其他菜单项相比，图片与普通标签的对齐方式会不正确。这个构件不是专用于图片的；任何构件都可以放在未缩进的“图片”位置。通常，GtkPixmapMenuItem是用Gnome菜单创建函数隐含创建的。

35. GtkTearoffMenuItem

库：Gtk+

头文件：gtk/gtktearoffmenuitem.h

描述：GtkTearoffMenuItem是一个“穿孔”，代表一个点 - 在这里菜单可以被“拖开”（在一个顶级窗口中保持可见，便于用户访问）。缺省时，Gnome菜单都包含一个tearoff菜单，但是用户可以用Gnome控制中心完全禁用它们。

36. GtkListItem

库：Gtk+

头文件：gtk/gtklistitem.h

描述：GtkListItem是一个不可见的容器，它允许它的子构件显示在一个GtkList构件上。也就是，只有列表项能出现在列表中。

37. GtkTreeItem

库：Gtk+

头文件：gtk/gtktreeitem.h

描述：GtkTreeItem是一个不可见的容器，它允许子构件能显示在一个GtkTree上。只有树数据项能出现在树构件上。

38. GtkEventBox

库：Gtk+

头文件：gtk/gtkeventbox.h

描述：GtkEventBox构件也许是Gtk+中最简单的容器构件。它的唯一效果就是有一个GdkWindow窗口。某些操作只对有窗口的构件起作用（比如设置背景颜色或捕获事件）。如果想在无窗口的构件上执行这些操作，可以将构件放在一个事件盒中，然后在事件盒上执行需要的操作，能起到同样的效果。

39. GtkHandleBox

库：Gtk+

头文件：gtk/gtkhandlebox.h

描述：GtkHandlebox构件能够给它的子构件添加一个手柄。拖动手柄，子构件可以从窗口上移开，并放在用户桌面的其他地方。手柄盒一般用于工具条。GnomeDock和GnomeDockItem提供了一个更灵活的(但是也更精巧)选择。

40. GtkScrolledWindow

库：Gtk+

头文件：gtk/gtkscrolledwindow.h

描述：GtkScrolledWindow为它的子构件提供了一个水平和垂直的滚动条。可选状态，当整个子构件都可见时，滚动条可以隐藏起来。如果子构件在它的GtkWidgetClass中有set_scroll_adjustments_signal，滚动窗口把它们当作滚动条的调整值。否则，滚动窗口用一个GtkViewport 构件滚动整个构件。(例如GtkCList；列标题不会滚动，只有列表内容滚动。如此GtkCList提供了一个滚动调整信号。)

41. GtkViewport

库：Gtk+

头文件：gtk/gtkviewport.h

描述：GtkViewport或多或少是GtkScrolledWindow构件的一个实现细节。它包含了一个不能提供set_scroll_adjustments_signal 方法的构件，为子构件提供这样一个信号。要了解更多信息，参看GtkScrolledWindow。

42. GtkBox

库：Gtk+

头文件：gtk/gtkbox.h

描述：GtkBox是GtkVBox、GtkHBox和GtkButtonBox的抽象基类。它是不可见的布局容器。

43. GtkHBox

库：Gtk+

头文件：gtk/gtkhbox.h

描述：GtkHBox是一个GtkBox，将子构件从左往右组装。左边看作组装盒的开始。

44. GnomeAppBar

库：Gnome

头文件：libgnomeui/gnome-appbar.h

描述：GnomeAppBar是一个简单的状态条，带一个可选的进度条。它不像GtkStatusbar有一个“上下文”。

45. GnomeDateEdit

库：Gnome

头文件：libgnomeui/gnome-dateedit.h

描述：GnomeDateEdit允许用户编辑日期和时间。如果仅仅对日期感兴趣，时间编辑部分可以关掉。

46. GtkCombo

库：Gtk+

头文件：gtk/gtkcombo.h

描述：GtkCombo是一个带下拉菜单“快速选择”的文本输入框。如果想限制用户只能选择一些固定的选项，GtkOptionMenu更合适。GtkCombo允许用户输入一些东西，但是还提供了一些建议值。GnomeEntry是一个类似于GtkCombo的组合框。它将用户输入的字符串添加到下拉列表中，并且在会话与会话之间能自动记住列表。

47. GnomeEntry

库：Gnome

头文件：libgnomeui/gnome-entry.h

描述：GnomeEntry是GtkCombo构件的扩展，将下拉菜单作为历史选项。如果用户输入一些在历史列表中还没有的东西，GnomeEntry将它添加进去，并保存在一个配置文件中，在应用程序下次启动时，能够再次加载。

48. GnomeFileEntry

库：Gnome

头文件：libgnomeui/gnome-file-entry.h

描述：GnomeFileEntry是一个GnomeEntry，将文件名保存在历史记录中。它有一个“浏览”按钮，点击时会弹出一个GtkFileSelection。它还有一个“只浏览目录”的模式。

49. GnomeNumberEntry

库：Gnome

头文件：libgnomeui/gnome-number-entry.h

描述：GnomeNumberEntry允许用户输入一个数字，它在它的下拉菜单中保存了一个曾经输入数字的历史记录。它还有一个“计算器”按钮，点击时会弹出一个GnomeCalculator。

50. GnomeProcBar

库：Gnome

头文件：libgnomeui/gnome-procbar.h

描述：GnomeProcBar用在Gnome面板小程序上，用来显示CPU和内存负载，还用在Gtop程序中(Gtop是top程序的图形化克隆)。它显示一个可变长度的着色长条，它还能用于显示任何类型的经常变化的值。

51. GtkStatusbar

库：Gtk+

头文件：gtk/gtk.h

描述：GtkStatusbar是一个状态条构件，它在窗口的底部显示一行文本。

52. GtkVBox

库：Gtk+

头文件：gtk/gtkvbox.h

描述：GtkVBox是一个GtkBox，将构件从顶部向底部组装。顶部被看作组装盒的“开始”。

53. GnomeCalculator

库：Gnome

头文件：libgnomeui/gnome-calculator.h

描述：GnomeCalculator是作为一个GtkWidget实现的简单计算器。

54. GnomeGuru

库：Gnome

头文件：libgnomeui/gnome-guru.h

描述：GnomeGuru试图实现一个“向导”构件（一系列的页面，代表用户能一步步完成的任务中某些步骤）。它还不成熟，不应该使用它。现在有一个新的构件，称为 GnomeDruid，可能会在 Gnome 的下一个版本中取代 GnomeGuru。在这之前，GnomeDruid 极有可能作为一个外接程序模块提供给用户，所以如果想要一个向导构件，可以找一下 GnomeDruid 构件。

55. GnomeIconEntry

库：Gnome

头文件：libgnomeui/gnome-icon-entry.h

描述：GnomeIconEntry 与 GnomeColorPicker 和 GnomeFontPicker 很相似。它是一个显示当前选中图标的按钮，还有一个图标浏览按钮，点击它可以设置新图标。它过去有一个文本输入框，用来输入图标文件名，所以它被称为 GnomeIconEntry 而不是 GnomeIconPicker。

56. GnomeIconSelection

库：Gnome

头文件：libgnomeui/gnome-icon-sel.h

描述：GnomeIconSelection 浏览图标文件，它用在 GnomeIconEntry 中，但是也可以直接使用。

57. GnomeLess

库：Gnome

头文件：libgnomeui/gnome-less.h

描述：GnomeLess 是 GtkText 构件的简单扩展，从一个文件或文件描述符加载一个文件并显示它。要尽量避免使用这个构件，因为它不是特别有用，所以它可能会从今后的 Gnome 版本中消失。

58. GnomePaperSelector

库：Gnome

头文件：libgnomeui/gnome-paper-selector.h

描述：GnomePaperSelector 是另一个要避免使用的构件，在 Gnome 1.0 发布版本中，它是严格的实验性构件。它允许用户选择打印纸张尺寸。

59. GnomePixmapEntry

库：Gnome

头文件：libgnomeui/gnome-pixmap-entry.h

描述：GnomePixmapEntry 与 GnomeIconEntry 构件的目的完全一样，它允许用户选择一个图片。喜欢某一个或另一个构件的唯一理由是 GnomeIconEntry 将图象按比例缩小到标准的 Gnome 图标大小。

60. GnomeSpell

库：Gnome

头文件：libgnomeui/gnome-spell.h

描述：GnomeSpell 是一个拼写检查接口，它在内部使用 ispell 程序。这个构件应该看作实

验性的，要避免在程序中使用。

61. GtkColorSelection

库：Gtk+

头文件：gtk/gtkcolorsel.h

描述：GtkColorSelection允许用户指定颜色，用一个轮形或滑块调色板来指定一种颜色。

它构成了GtkColorSelectionDialog的内容。

62. GtkGammaCurve

库：Gtk+

头文件：gtk/gtkgamma.h

描述：GtkGammaCurve 允许用户编辑一个曲线，它是专门用于 GIMP的构件。很少有应用程序会用到它。

63. GtkButtonBox

库：Gtk+

头文件：gtk/gtkbbox.h

描述：GtkButtonBox是一种特殊类型的GtkBox，用于放置对话框的按钮。它有水平和垂直两种变体。GnomeDialog会创建一个按钮盒，所以在Gnome下编程时，不需要直接使用这个构件。

64. GtkHButtonBox

库：Gtk+

头文件：gtk/gtkhbbox.h

描述：GtkHButtonBox是GtkButtonBox的水平变体。

65. GtkVButtonBox

库：Gtk+

头文件：gtk/gtk.h

描述：GtkVButtonBox是GtkButtonBox的垂直变体。

66. GtkLayout

库：Gtk+

头文件：gtk/gtklayout.h

描述：GtkLayout是一个容器，给人的感觉好象有无限的尺寸似的。因为X窗口最大可以是32 768像素(子构件必须放在父构件的X窗口上)。其他的构件都用简单移动它们的GdkWindow窗口来滚动。GtkLayout比这些构件更聪明些。

67. GnomeCanvas

库：Gnome

头文件：libgnomeui/gnome-canvas.h

描述：它绘制不闪烁的结构化图形，对定制的显示也是非常理想的。

68. GnomeIconList

库：Gnome

头文件：libgnomeui/gnome-icon-list.h

描述：GnomeIconList用在Gnome文件管理器。它显示图标和它们的名称，用户能够通过

拖动鼠标选择图标组。

69. GnomeDockBand

库：Gnome

头文件：libgnomeui/gnome-dock-band.h

描述：GnomeDockBand包含一行或一列GnomeDockItem。GnomeDock也包含了一个或多个GnomeDockBand。

70. GnomeDock

库：Gnome

头文件：libgnomeui/gnome-dock.h

描述：GnomeDock里面包含GnomeDockBand，而GnomeDockBand里面包含GnomeDockItem。GnomeDock允许用户将工具条和应用程序的其他部件重新定位。

71. GtkCList

库：Gtk+

头文件：gtk/gtkclist.h

描述：GtkCList是一个多列的列表构件；它也是GtkCTree的基类。GtkCList在每个单元格中显示文本和/或图片；单元格里不能容纳子构件。GtkCList仅仅是一个容器，因为它使用构件作为它的栏标题。你可能更愿意使用GtkList，因为在它的列表项中能够使用子构件，但是它效率较低，列表项最大只能有32 768个像素。

72. GtkCTree

库：Gtk+

头文件：gtk/gtkctree.h

描述：GtkCTree与GtkCList类似，但是显示一个可展开的节点而不是显示一个简单的列表。GtkTree是更灵活的树构件（在树单元里可以是任意类型的构件），但是它效率较低，只能有32 768个像素大小。

73. GtkFixed

库：Gtk+

头文件：gtk/gtkfixed.h

描述：GtkFixed容器允许用户将子构件放在绝对坐标上，并且给子构件所需要的尺寸大小。最好还是用其他的布局容器构件。

74. GtkNotebook

库：Gtk+

头文件：gtk/gtknotebook.h

描述：笔记本构件为用户显示“多页”内容。用户可以通过选择它的“页标签”在“页”间移动。每个添加到GtkNotebook上的子构件都成为一页，还可以在页标签上使用其他的构件。通常具有一行以上的页标签是一种很糟糕的做法，但是它确实允许这么做。还可以将标签放在构件的左边、右边或下边，但是为界面连贯性起见，最好还是将它放在构件的顶部。

75. GtkFontSelection

库：Gtk+

头文件：gtk/gtkfontsel.h

描述：GtkFontSelection是一个组合构件，允许用户从系统可用的字体中选择一种字体。GtkFontSelectionDialog里面包含了GtkFontSelection的一个实例。Gnome应用程序应该在一个GnomeDialog中放一个GtkFontSelection构件，而不是使用GtkFontSelectionDialog构件。

76. GtkPaned

库：Gtk+

头文件：gtk/gtkpaned.h

描述：GtkPaned构件将一个区域划分为两个用户可调整大小的部分。它有水平和垂直两个变体。

77. GtkHPaned

库：Gtk+

头文件：gtk/gtkhpaned.h

描述：GtkHPaned是GtkPaned的水平变体，它将一个区域划分为左右两个区域。

78. GtkVPaned

库：Gtk+

头文件：gtk/gtkvpaned.h

描述：GtkVPaned是GtkPaned构件的垂直变体，它将一个区域划分为上下两个部分。

79. GtkList

库：Gtk+

头文件：gtk/gtklist.h

描述：GtkList显示一个项目列表。每个项目称为一个GtkListItem，GtkListItem是一个可以包含任何种类构件的容器。GtkList的大小是有限的，因为它的列表项是放在列表的GdkWindow中的固定坐标上的，滚动是通过移动GdkWindow做到的。GdkWindow最大可以是32 768像素，任何超过这个范围的列表项都是不可见的。GtkCList构件克服了这个限制，但是不能包含任意种类的构件。

80. GtkMenuShell

库：Gtk+

头文件：gtk/gtkmenushell.h

描述：GtkMenuShell是一个内部包含GtkMenuItem的构件的抽象基类。它的两个子类是GtkMenu和GtkMenuBar。

81. GtkMenuBar

库：Gtk+

头文件：gtk/gtkmenubar.h

描述：GtkMenuBar是一个菜单条。它包含一个或多个菜单项，通常，每个菜单项都包含一个子菜单（也就是，一个GtkMenu带多个菜单项）。例如，菜单条也许有一个菜单项叫“文件”，该菜单项包含子菜单项“打开”和“退出”。

82. GtkMenu

库：Gtk+

头文件：gtk/gtkmenu.h

描述：GtkMenu包含菜单项。因为它不会由程序员显示出来（用Ggtk_widget_show()），所

以它是独一无二的。菜单会响应用户的动作弹出来。

83. GtkPacker

库：Gtk+

头文件：gtk/gtkpacker.h

描述：GtkPacker是一个布局容器构件，是受到 Tk工具包启发而创建的。如果你对 Tk工具包很熟悉，就会发现它比标准的 Gtk+布局容器更容易使用。

84. GtkSocket

库：Gtk+

头文件：gtk/gtksocket.h

描述：GtkSocket是在一个应用程序中的“洞”，它允许来自与另一个应用程序的 GtkPlug构件嵌入到里面。

85. GtkTable

库：Gtk+

头文件：gtk/gtktable.h

描述：GtkTable是最重要的 Gtk+布局构件之一。

86. GtkTed

库：Gnome

头文件：libgnomeui/gtk-ted.h

描述：GtkTed是Gnome早期遗留物。“ted”意为“表格编辑器”。它是一种原始的 GUI生成器。现在至少有两个 GUI生成器项目 (Glade和GLE)正在开发，所以可以忽略这个构件，它也会在libgnomeui库今后的版本中消失。

87. GtkToolbar

库：Gtk+

头文件：gtk/gtktoolbar.h

描述：GtkToolbar是一个工具条构件。通常 Gnome应用程序应该使用 Gnome的helper函数而不是直接创建工具条。

88. GtkTree

库：Gtk+

头文件：gtk/gtktree.h

描述：GtkTree与GtkCTree的关系和GtkList与GtkCList的关系一样。也就是，GtkTree比GtkCTree更灵活(tree items可以包含任何构件)但是也比GtkCTree更慢，只能容纳有限数量的项。Item的最大数目依赖于行高，但总的行高必须能容纳在一个 GdkWindow内，也就是32 768个像素。

89. GnomeAnimator

库：Gnome

头文件：libgnomeui/gnome-animator.h

描述：GnomeAnimator显示一系列图像，生成一个动画。它有一个“循环模式”和一个“播放一次”模式。GnomeAnimator API在Gnome 1.0中标志为“不成熟”，在未来的版本中可能会变得不兼容；它是一个实验性的构件。

90. GnomePixmap

库：Gnome

头文件：libgnomeui/gnome-pixmap.h

描述：GnomePixmap应该是比GtkPixmap更好的构件，虽然它与后者有同样的作用。

GnomePixmap在视觉上更灵活。它还使用Imlib可以加载多种不同格式的图像；所以使用它更方便。

91. GnomeStock

库：Gnome

头文件：libgnomeui/gnome-stock.h

描述：GnomeStock是一个封装了GnomePixmap的容器。它自动地创建pixmap的“不敏感”和“具有焦点”拷贝，以反映构件的状态。它可以设置为一个Gnome内置的pixmap宏，比如GNOME_STOCK_PIXMAP_CUT(小剪刀图片)，GNOME_STOCK_PIXMAP_PRINT(一个小打印机)等。还可以在运行时注册一个新的内置的与应用程序图片。Gnome在用GnomeUIInfo创建菜单和工具条时，在内部使用这个构件。

92. GtkMisc

库：Gtk+

头文件：gtk/gtkmisc.h

描述：GtkMisc抽象基类用来设置它的子类的“对齐值”和“填充量”。对齐值是介于0.0和1.0之间的浮点数，0.0是左对齐，1.0是右对齐，0.5是居中对齐。只有在GtkMisc构件接收到比它所请求更多的尺寸分配时才管用。对齐值将构件的自然约束框在它实际约束框中定位。填充量是增加到构件尺寸请求中的像素值。构件让这些像素值的地方空着。

不是从GtkMisc中派生的构件也可以变成“可对齐的”，只需将构件放在一个GtkAlignment容器中就可以了。

93. GtkLabel

库：Gtk+

头文件：gtk/gtklabel.h

描述：GtkLabel简单地显示一个文本串。如果文本串包含新行，GtkLabel显示多行。GtkLabel标签从它的父类(GtkMisc)中取得对齐参数，可以用这个参数让文本居中，或将它向左或向右移动。对齐与版面调整不一样。版面对齐根据彼此的内容调整多行文本的位置。版面左对齐意味着每一行从同样的位置开始，版面右对齐意味着每一行都在同样的地方终止，版面居中对齐意味着每一行都以一条假象的线居中对齐。版面对齐对只有一行的标签没有任何意义。对齐用gtk_misc_set_alignment()函数设置整块文本在所分配的空间中的位置。对齐只在标签接收到比它请求更多的空间时才起作用(它请求足够的空间以容纳文本块)。GtkLabel标签是GTK_NO_WINDOW类型的构件(这意味着它们不会接收到事件，直接绘制在父构件的背景上)。

94. GtkAccelLabel

库：Gtk+

头文件：gtk/gtkaccellabel.h

描述：GtkAccelLabel与其他构件相关联，作为标签的一部分，为该构件显示加速键。

95. GtkClock

库：Gnome

头文件：libgnomeui/gtk-clock.h

描述：GtkClock是一个标签，用来显示时间。它可以周期性地更新时间（就像时钟一样！）。

96. GtkTipsQuery

库：Gtk+

头文件：gtk/gtktipsquery.h

描述：GtkTipsQuery是一个显示工具提示的标签。它还有一个“ What 's This? ”功能。调用gtk_tips_query_start()函数切换到“ 查询模式 ”。在查询模式中，光标变成一个问号；当鼠标在应用程序的构件上面移动时，GtkTipsQuery显示这些构件的工具提示。如果用户点击构件，GtkTipsQuery引发一个“ widget_selected ”信号，可以用它来为构件显示更广泛的帮助信息。可以在tkTooltip的“ 私有 ”组件中存储更广泛的帮助信息，或用 gtk_object_set_data()函数来存储一些应用程序相关的信息。

97. GtkArrow

库：Gtk+

头文件：gtk/gtkarrow.h

描述：GtkArrow显示一个箭头。它是一种简单的 GTK_NO_WINDOW构件不能接受事件。在缺省主题中，箭头仅仅是一个三角形。

98. GtkImage

库：Gtk+

头文件：gtk/gtkimage.h

描述：GtkImage在一个构件中显示一个GdkImage。它只在已经有了一个GdkImage时才有用；要显示一个确定的图像，应该使用 GnomePixmap。

99. GtkPixmap

库：Gtk+

头文件：gtk/gtkpixmap.h

描述：GtkPixmap显示一个GdkPixmap。如果正在使用Gnome，建议使用GnomePixmap构件。理由在GnomePixmap的介绍中已有说明。

100. GtkCalendar

库：Gtk+

头文件：gtk/gtkcalendar.h

描述：GtkCalendar显示一个日历页(一个月)。它允许用户选择一个日期。GnomeDateEdit构件在弹出菜单中使用GtkCalendar构件。

101. GtkDrawingArea

库：Gtk+

头文件：gtk/gtkdrawingarea.h

描述：GtkDrawingArea是GdkWindow的一个“ 瘦 ”封装。它提供一个空白区域，可用于绘画。正常情况，应该连接到它的“ configure_event ”信号，以捕获区域的大小变化，在一个

“expose_event”信号处理程序中实现绘画。要消除闪烁，可以保留一个与绘画区尺寸相等的GdkPixmap，画到GdkPixmap上。在“expose_event”处理程序中，简单将暴露的区域从GdkPixmap复制到绘画区。对高级图形绘画，GnomeCanvas可能是更好的选择。

102. GtkCurve

库：Gtk+

头文件：gtk/gtkcurve.h

描述：GtkCurve用于在GtkGammaCurve构件中显示曲线。它是一种带曲线会话能力的绘图区扩展。不大可能会用到这个构件。

103. GtkDial

库：Gtk+

头文件：gtk/gtkdial.h

描述：GtkDial是一种“里程计”显示。缺省时，用户可以拖动指针旋转，改变标度盘的值。GtkDial还有一种“仅供观察”模式。

104. GtkEditable

库：Gtk+

头文件：gtk/gtkeditable.h

描述：GtkEditable是允许用户编辑文本的构件的抽象基类。基类接口允许光标定位，获取字符等待。它还包含一个“changed”信号，可用于监测用户的输入。

105. GtkEntry

库：Gtk+

头文件：gtk/gtkentry.h

描述：GtkEntry允许用户输入一行文本。它有一种“口令”模式，在“口令”模式中，输入框中的文本被星号“*”替代。

106. GtkSpinButton

库：Gtk+

头文件：gtk/gtkspinbutton.h

描述：GtkSpinButton是一个定制的GtkEntry，允许用户输入一个数字。Spin buttons给输入框增加了向上和向下两个按钮，用户可以在可能取值之间快速调整。

107. GtkText

库：Gtk+

头文件：gtk/gtktext.h

描述：GtkText是一个文本构件。它能显示文本，并且还提供简单的文本编辑实用程序。

108. GtkRuler

库：Gtk+

头文件：gtk/gtkruler.h

描述：GtkRuler是水平标尺和垂直标尺构件的抽象基类。标尺构件在GIMP中用于显示图象的尺寸。

109. GtkHRuler

库：Gtk+

头文件：gtk/gtkhruler.h

描述：GtkRuler的水平变体。

110. GtkVRuler

库：Gtk+

头文件：gtk/gtkvruler.h

描述：GtkRuler的垂直变体。

111. GtkRange

库：Gtk+

头文件：gtk/gtkrange.h

描述：GtkRange是“滑块类”构件的一个抽象基类。这些构件当“滑块”在“滑槽”内移动时修改某些数字值。它的两个子类是 GtkScale，用于让用户输入一个数值，和 GtkScrollbar，Gtk+的滚动条构件。

112. GtkScale

库：Gtk+

头文件：gtk/gtkscale.h

描述：GtkScale允许用户通过移动一个滑块来输入数字值。它能在滑块的上面显示当前值；如果精确值没有什么关系，可以将这个特性关闭，或者，也可以以其他方法提供反馈。

GtkScale是一个抽象基类，必须使用它的垂直或水平变体。

113. GtkHScale

库：Gtk+

头文件：gtk/gtkhscale.h

描述：GtkHScale是GtkScale的水平变体。

114. GtkVScale

库：Gtk+

头文件：gtk/gtkvscale.h

描述：GtkVScale是垂直的GtkScale。

115. GtkScrollbar

库：Gtk+

头文件：gtk/gtkscrollbar.h

描述：GtkScrollbar是一个抽象基类，为垂直和水平滚动条提供公用接口。

116. GtkHScrollbar

库：Gtk+

头文件：gtk/gtkhscrollbar.h

描述：水平滚动条。

117. GtkVScrollbar

库：Gtk+

头文件：gtk/gtkvscrollbar.h

描述：垂直滚动条。

118. GtkSeparator

库：Gtk+

头文件：gtk/gtkseparator.h

描述：GtkSeparator构件是一分隔线，可以让用户界面更有吸引力。例如，用GtkHSeparator将GnomeDialog对话框的内容区与它的按钮分隔开。

119. GtkHSeparator

库：Gtk+

头文件：gtk/gtkhseparator.h

描述：水平的GtkSeparator。

120. GtkVSeparator

库：Gtk+

头文件：gtk/gtkvseparator.h

描述：垂直的tkSeparator。

121. GtkPreview

库：Gtk+

头文件：gtk/gtkpreview.h

描述：GtkPreview显示一个RGB图像，GIMP用它以显示一个图像转换的预览效果。

122. GtkProgress

库：Gtk+

头文件：gtk/gtkprogress.h

描述：GtkProgress是进度显示的抽象基类。在 Gtk+ 1.2 中它只有一个具体的子类 (GtkProgressBar)。未来的Gtk+版本也许会增加其他的进度构件。

123. GtkProgressBar

库：Gtk+

头文件：gtk/gtkprogressbar.h

描述：GtkProgressBar是一个灵活的进度条构件。除进度条功能以外，还可以在上面显示文本，还有一个“活动”模式以指示“正在活动，但不知道任务的大小”。在“活动”模式中，一个小滑块会前后移动。GtkProgressBar是可配置的，为了与其他应用程序保持一致，应该试一下它的缺省外观和感觉。

124. ZvtTerm

库：Zvt

头文件：zvt/zvtterm.h

描述：ZvtTerm是随gnome-libs发布的，但是它并不在 libgnomeui中。它是一个单独的libzvt库。ZvtTerm只是一个终端仿真器；可以产生一个子进程，在构件内运行，并与用户交互。ZvtTerm提供与Gnome桌面环境中的gnome-terminal程序具有的全部功能。

C.3 画布项

1. GnomeCanvasItem

库：Gnome

头文件：libgnomeui/gnome-canvas.h

描述：GnomeCanvasItem是画布项的抽象基类。

2. GnomeCanvasRE

库：Gnome

头文件：libgnomeui/gnome-canvas-rect-ellipse.h

描述：GnomeCanvasRE是矩形和椭圆画布项的抽象基类。今后，它还会是GnomeCanvasArc项的基类。

3. GnomeCanvasEllipse

库：Gnome

头文件：libgnomeui/gnome-canvas-rect-ellipse.h

描述：GnomeCanvasEllipse在画布上绘制一个椭圆。

4. GnomeCanvasRect

库：Gnome

头文件：libgnomeui/gnome.h

描述：GnomeCanvasRect在画布上绘制一个矩形。

5. GnomeCanvasGroup

库：Gnome

头文件：libgnomeui/gnome-canvas.h

描述：GnomeCanvasGroup是一个包含其他GnomeCanvasItem的GnomeCanvasItem。它创建一个GnomeCanvas中画布项的分层次树状结构。GnomeCanvas创建一个成为“根”的特殊的GnomeCanvasGroup，所有用户创建的画布项加到“根”组。

6. GnomeCanvasImage

库：Gnome

头文件：libgnomeui/gnome-canvas-image.h

描述：GnomeCanvasImage在画布上显示一幅图片(更明确地说，是一幅GdkImLibImage)。

7. GnomeCanvasLine

库：Gnome

头文件：libgnomeui/gnome.h

描述：GnomeCanvasLine在画布上显示一条线段或一系列的线段；它还能将最后一根线的终点与第一根线的起点连接起来，显示一个未填充的多边形。在线的两端还可以带箭头。

8. GnomeCanvasPolygon

库：Gnome

头文件：libgnomeui/gnome-canvas-polygon.h

描述：GnomeCanvasPolygon显示一个填充的多边形。用GnomeCanvasLine画中空的多边形。

9. GnomeCanvasText

库：Gnome

头文件

libgnomeui/gnome-canvas-text.h

描述：GnomeCanvasText在画布上显示一些文本。

10. GnomeCanvasWidget

库：Gnome

头文件：libgnomeui/gnome-canvas-widget.h

描述：GnomeCanvasWidget模拟一个GtkContainer构件。它容纳一个子构件，并将它显示在画布上。

11. GnomeCanvasTextItem

库：Gnome

头文件：libgnomeui/gnome-icon-item.h

描述：GnomeCanvasTextItem用于GnomeIconList构件内部。不应该直接使用它，它被当作一个实现细节，在今后 Gnome库的版本中可能会变化。

C.4 其他对象

1. GnomeClient

库：Gnome

头文件：libgnomeui/gnome-client.h

描述：GnomeClient是一个GtkObject对象，它隐蔽了会话管理的细节，并为 Gnome应用程序提供了很好的会话管理 API。

2. GnomeDEntryEdit

库：Gnome

头文件：libgnomeui/gnome-dentry-edit.h

描述：GnomeDEntryEdit是一个非常特别的对象。它是一种“构件管理器”，它创建并监视两个子构件：一个“简单”的页面，一个“高级”的页面。这两页结合起来让用户编辑一个Gnome的.desktop文件。GnomeDEntryEdit倾向于与已有的GtkNotebook联合使用。它本身并不是GtkNotebook的子类，因为允许会在GnomePropertyBox中使用GtkBoteBook构件。Gnome面板和Gnome菜单编辑器使用这个构件。

3. GnomeDockLayout

库：Gnome

头文件：libgnomeui/gnome-dock-layout.h

描述：GnomeDockLayout维护在GnomeDock中的构件的当前位置信息。它能够加载和保存这个信息，GnomeDock使用这个功能来保存和加载工具条的位置。GnomeApp使用GnomeDock构件来排列它的布局。

4. GnomeMDIChild

库：Gnome

头文件：libgnomeui/gnome-mdi-child.h

描述：GnomeMDIChild是一个抽象接口。必须派生新的子类，或者为了利用GnomeMDI对象，使用GnomeMDIGenericChild构件。GnomeMDI是一个“多文档接口”管理器。

5. GnomeMDIGenericChild

库：Gnome

头文件：libgnomeui/gnome-mdi-generic-child.h

描述：GnomeMDIChild构件的一般性的实现。对较复杂的应用程序，也许需要重新写一个定制的GnomeMDIChild实现。

6. GnomeMDI

库：Gnome

头文件：libgnomeui/gnome-mdi.h

描述：GnomeMDI跟踪多个文档。用户能够配置如何在应用程序中排列文档；它们也许被放在同一个GtkWindow中的GtkNotebook上，或者每个文档都有自己的GtkWindow窗口。还有，用户也能将文档（“笔记本页”）拖出GtkWindow，它们会被放置在自己的顶级窗口中。

7. GtkData

库：Gtk+

头文件：gtk/gtkdata.h

描述：GtkData是可为多个对象所共享的数据片断的抽象基类。现在它的接口还是空的，今后，对所有GtkData对象，也许会有一些通用的操作函数。

8. GtkAdjustment

库：Gtk+

头文件：gtk/gtkadjustment.h

描述：GtkAdjustment代表一个数值。它还存储一个最大值和一个最小值，一个“单步增量”，一个“单页增量”和一个“页面值”。一些对象并不使用在GtkAdjustment中的所有的这些值，一些对象对这些值的解释也略有不同。GtkRange构件(包括它的GtkScale和GtkScrollbar子类)允许用户将滑块在最大值和最小值之间移动。点击GtkScrollbar构件端部的箭头让滚动条推动一个“单步增量”，用鼠标点击它们会将滚动条移动一个“单页增量”。“页面值”决定滚动条滑块的尺寸大小(就是在整个范围内的同一单元上可见页的长度)。当GtkAdjustment的值改变时，引发一些信号。

9. GtkTooltips

库：Gtk+

头文件：gtk/gtktooltips.h

描述：GtkTooltips将构件与一些帮助文本联系起来。如果将鼠标指针在构件上方做短暂停留，工具提示会出现，显示帮助文本。GtkTooltips也能存储一些“私有”文本；可以将它与GtkTipsQuery一起使用来显示扩展的帮助信息。

10. GtkItemFactory

库：Gtk+

头文件：gtk/gtkitemfactory.h

描述：GtkItemFactory用于简化菜单创建过程，Gnome应该使用GnomeUIInfo模板来创建菜单。

